KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Composable and Embeddable Software Design for Robotic and Cyber-Physical Systems

Lin Zhang

Supervisors:
Prof. dr. ir. H. Bruyninckx
Prof. dr. ir. P. Slaets, co-supervisor

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Mechanical Engineering

March 2017

Composable and Embeddable Software Design for Robotic and Cyber-Physical Systems

Lin ZHANG

Examination committee:
Prof. dr. ir. W. Sansen, chair
Prof. dr. ir. H. Bruyninckx, supervisor
Prof. dr. ir. P. Slaets, co-supervisor
Prof. dr. ir. J. Swevers
Prof. dr. ir. J. De Schutter
Prof. dr. ir. G. Deconinck
Prof. dr. ir. M.J.G. van de Molengraft
(Technische Universiteit Eindhoven)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Mechanical Engineering

March 2017

Preface

This dissertation would not have been finalized without the help and support of many individuals who have greatly assisted me in my PhD years. Undertaking this PhD has been a truly challenging and rewarding experience, I would like to express my gratitude to all those who have made this dissertation possible.

My sincerest gratitude goes first and foremost to my supervisor, Prof. Herman Bruyninckx, who has supported me with his great guidance and encouragement on the journey of pursuing my PhD. I really appreciate all his effort and patience that helped me to get into the domain of robotics and systematic modelling, especially, I would like to thank him for providing the opportunities to sharpen my programming, writing and teaching skills, as well as to broaden my scientific and academic horizons and interests. His enthusiasm has deeply influenced my attitude towards scientific research, it is my great honor to be mentored by him.

I would like to address my great acknowledgement to my co-supervisor, Prof. Peter Slaets, for his constant support throughout my doctoral research. I would also like to thank Peter for bringing the opportunity of pursuing this PhD and working with a group of friendly and knowledgeable people.

I would like to thank the members of my assessor committee and jury, Prof. Joris De Schutter, Prof. Jan Swevers, Prof. Geert Deconinck and Prof. René van de Molengraft for providing valuable feedback to the research and the dissertation. I would also like to thank Prof. Willy Sansen, who I admired since I started studying electronics many years ago, for chairing my defence.

I want to thank my former colleagues for having helped with almost everything that was important at the beginning of my PhD. I would like to thank Tinne De Laet for getting me into the open-source world and for shortening the detours as a Linux user; thank you Markus Klotzbücher for polishing and updating my skills and knowledge on embedded systems as well as all the valuable guidance and instructions on programming; I want to thank Dominick Vanthienen and Steven Bellens for organizing the OROCOS seminar, Hans Wambacq for helping

me to review lectures on motor driving, and Enrico Di Lello for the practical information about the gym; I would also like to thank Azamat Shakhimardanov for joining the PhD school trips and his interests and suggestions on the gadgets I built.

I am very grateful to Nico Hübel for pointing me in the right direction at the right time, I appreciate all his mental and scientific support and help. Together, I would like to thank Enea Scioni and Johan Philips for the crucial and inspirational discussions on software modelling and implementation, Sebastian Blumenthal and Sven Schneider for sharing their system design experiences, and my special thanks go to Johan for translating the abstract to Dutch.

I would like to thank Wilm for organizing non-scientific discussions in the office (after work of course) and the interesting 3D vision sensor test together with Maxim and Enea; thank you Jonas and Bart for enriching my vision with your impressive results on exoskeleton and drones; thank you Kevin for dressing up as St. Nicholas and the gifts every year; thank you Cristian for sharing the travelling experience; and thank you Yuhda for joining the chicken legs Sundays.

I want to extend my appreciation to the colleagues working at the electronic workshop and Fab-Lab, Bertram, Jean-Pierre, Thomas and Marc for their valuable advice and suggestions; I would like to thank the secretary group, Karin, Anja, Regine, Lieve, Marijke, Valérie and Marina, for helping me with all kinds of administrative issues; and I would also like to thank Jan and Ronny for their support on the IT problems.

I would like to thank Diego Pérez Losada and Yan Zhou, who helped me to improve my coding skills and to inspire the implementation of the software in this dissertation.

My special thanks go to my friends at GroupT, Luc Bienstman, Johan Mannaerts and Wim Polet for their long-standing support and help, and I would like to thank Luc Geurts who helped me to redirect myself towards the PhD.

I am grateful to my Chinese friends in Belgium, Yansong, Ye, Yuchen, Yunhao, Zhibin, Jian, Qianyong, Hua and Liqun for the warm friendship which made me feel at home. I would also like to thank Min and Haibin for organizing wonderful dinners and their hospitalities. I also enjoyed the cheerful moments with Jun Qian, Xiaoming, Jingyu, Jun and Xinling, Han, Hao, Cheng, Fei, Dejian, Wei, Xin and Marnix, Xin Lin, Yelin, Guoying, Rong, Sida, Chunxiao, Ke and Feng. I want to thank Xu, Liang and Yang who work in the office next door, reminding me that we can be productive during weekends as well.

I owe my deep appreciation to Prof. Hengjun Zhu (BJTU) for his great help and assistance during the past years. I would also like to thank Prof. Yan'an

Yao (BJTU) for his kind support and encouragement.

Most importantly, I want to sincerely thank my parents, Guiwang Zhang and Xiaoxia Zhao for their tireless encouragement, and I want to thank my parents-in-law, Zhijun Peng and Jun Hou, for their continuous understanding and support. My sincere acknowledgement also goes to my other family members for their great contribution in my life.

Finally, Yue Peng, my dear love, I owe the deepest gratitude to you, for the long march of being apart from each other since six years ago. Thank you for all the efforts, understanding, love and sacrifice throughout my studies. I am truly grateful to your dedication to the family and meticulous care to our lovely son Hengyu. Without your support, I could not have completed this journey.

Lin Zhang,
Heverlee, March 2017.

Abstract

System-of-systems is a multidisciplinary area that involves system integration as a key to address complex tasks or problems, usually by means of composing multiple *independently* controlled systems together as part of a large application that often *exists only temporarily*. The focus of this work is the cyber-physical system, which is, a class of system-of-systems in which computers and networks are organized to monitor and control physical processes.

This research presents a systematic methodology for the realization of autonomous coordination and self-reconfiguration on cyber-physical systems and system-of-systems, with concrete examples from robotics domain. The work models systems in a well-structured way using a set of system design patterns to improve the composability, flexibility and reusability of sub-systems, and it also explains how to integrate individual sub-systems as system-of-systems with predictable behaviours.

As its first contribution, the *Cyber-Physical Stack* (CPS) meta model presented in this work facilitates modelling structures of systems by explicitly describing *tasks*, *capabilities*, *functions* and *devices* in four layers as a *stack*. The CPS meta model connects the Composition Pattern, the Coordination-Configuration Pattern, and proposes a novel Computation Behaviour Composition Pattern, the latter composes the computation behaviour of a system by specifying the structural containment, connectivity of functions and data, plus the execution orders of the computations. The structure containment and connectivity are described by a computer readable *array-of-integer (AOI) representation* for software implementation.

As the second contribution, this dissertation improved the *Life-Cycle State Machine* (LCSM) by explicitly abstracting *resource* and *capability* as the two properties in the life-cycle of any sub-system. The LCSM is realized by means of a *composable Finite State Machine* (cFSM) meta model.

The third contribution of this dissertation is the *composable and embeddable*

software (CES), which is an efficient and highly reusable implementation for the CPS and cFSM meta models.

Featured in small scale, low cost and low power consumption, embedded platforms are essential in system-of-systems design, especially for educational activities. Therefore, the above mentioned meta models and their software implementation are made *embedded-friendly*, as *embeddability* is also a main focus in this research.

The fourth contribution is the best practices learned from cyber-physical system design using embedded devices, including ARM-based, microcontroller-based and FPGA-based platforms. The best practices cover the *optimizations* of resource consumption, run time efficiency and development efforts.

As the last contribution, this dissertation presents a complete design process using a concrete microcontroller-based setup as an example, to verify the validity and feasibility of the developed methodology and the software. This example provides a compact *teaching material* for system-of-systems design, one can simply reproduce the setup as the hardware list, the software code as well as the mechanical structure sketch are provided.

Beknopte samenvatting

"System-of-systems" (systemen die zelf uit onafhankelijke systemen bestaan), is een multidisciplinair onderzoeksdomein waarbij systeemintegratie de sleutel is tot het oplossen van complexe taken of problemen, vaak door zo'n compositie van meerdere onafhankelijk gecontroleerde systemen samen deel te laten uitmaken van een grote softwaretoepassing, die vaak slechts tijdelijk bestaat. Dit werk focust op cyber-fysische systemen, wat een speciale vorm is van "systems-of-systems" waarbij computers en netwerken georganiseerd zijn om fysische processen te observeren en controleren.

Dit onderzoek presenteert een systematische methodologie voor de realisatie van autonome coördinatie en zelfconfiguratie bij cyber-fysische systemen, met concrete voorbeelden binnen de robotica. De voorgestelde systeemmodellen zijn opgebouwd met systeemontwerppatronen wat de structuur ten goede komt en leidt tot verbeterde samenstelbaarheid, flexibiliteit en herbruikbaarheid van de subsystemen. Dit werk verklaart ook hoe individuele subsystemen geïntegreerd dienen te worden als systems-of-systems met voorspelbaar gedrag.

De eerste bijdrage is de ontwikkeling van het Cyber-Physical Stack (CPS) metamodel dat het modelleren van systeemstructuren vereenvoudigt door taken, bekwaamheden, functies en apparaten expliciet beschrijft in vier lagen, wat in vakjargon een "stack" genoemd wordt. Het CPS metamodel verbindt het Composition Pattern met het Coordination-Configuration Pattern en stelt een nieuw Computation Behaviour Composition Pattern voor, dat het berekeningsgedrag van een system samenstelt door het specificeren van de structurele omsluiting, connectiviteit van functies en data, én de uitvoervolgorde van de berekeningen. De structurele omsluiting en connectiviteit zijn beschreven door een array-of-integer (AOI) representatie die leesbaar is door een computer, namelijk meteen bruikbaar in een software-implementatie.

De tweede bijdrage is de verbetering van het Life-Cycle State Machine (LCSM) door expliciet bronnen en bekwaamheden als twee eigenschappen

in de levenscyclus van elk subsysteem te modelleren. Het LCSM is gerealiseerd door een samenstelbaar eindigetoestandsmachine (cFSM) metamodel.

De derde bijdrage is de samenstelbare en ingebedde software (CES), die een efficiënte en uiterst herbruikbare implementatie is voor de CPS en cFSM metamodellen.

Ingebedde platformen, kleine in grootte, laag in kosten en laag in energieverbruik, zijn essentieel in het ontwerp van system-of-systems, speciaal voor educatieve activiteiten. Daarom werden de vermeldde metamodellen en hun software-implementatie zo ontwikkeld dat ze eenvoudig ingebed kunnen worden en was dit ook de hoofdfocus van dit onderzoek.

De vierde bijdrage is de best practice die ontwikkeld werd door cyberfysische systemen te ontwerpen met ingebedde apparaten, inclusief ARM-gebaseerde, microcontroller-gebaseerde en FPGA-gebaseerde platformen. Deze best practice omvat de optimalisaties voor verbruik, efficiënte uitvoertijd en ontwikkelingstijd.

De laatste bijdrage is de voorstelling van een volledig ontwerpproces gebruikmakende van concrete microcontroller-gebaseerde opstellingen als voorbeeld, om de correctheid en haalbaarheid te verifiëren van de ontwikkelde methodologie en de software. Dit voorbeeld bevat compact didactisch materiaal voor systeemontwerp en men kan de opstelling eenvoudig reproduceren aangezien de lijst van onderdelen, de softwarecode en het mechanisch ontwerp beschikbaar gemaakt werden.

Abbreviations

AADL	Architecture Analysis and Design Language
ADC	Analog-to-Digital Converter
AOI	Array-of-integer
AXI	Advanced eXtensible Interface
BRAM	RAM block
CES	Composable and Embeddable Software
CFB	Computation function block
cFSM	composable Finite State Machine
CLB	Configurable Logic Blocks
CPB	Composite computation block
CPS	Cyber-Physical Stack
CST	Composite state
DSP	Digital Signal Processing
EDA	Electronic Design Automation
EP	Event Processing
EPT	External port
FF	Flip-flop
FPGA	Field Programmable Gate Array
GPIO	General Purpose Input/Output
HDL	Hardware Description Language
HLS	High Level Synthesis
I^2C	Inter-Integrated Circuit

IDE	Integrated Development Environment
IMU	Inertial Measurement Unit
IoT	Internet of Things
IP	Intellectual Property
IPT	Internal port
LCSM	Life Cycle State Machine
LST	Leaf state
LUT	Lookup table
MBE	Model Based Engineering
MDE	Model Driven Engineering
NFS	Network File System
OROCOS	Open RObot COntrol Software
PL	Programmable Logic
PS	Processing System
PWM	Pulse Width Modulation
ROS	Robot Operating System
SoC	System-on-Chip
SPI	Serial Peripheral Interface
Tcl	Tool command language
UART	Universal Asynchronous Receiver/Transmitter
VHDL	Very high speed integrated circuits Hardware Description Language

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis covers two major aspects: (1) a modelling methodology for autonomous *coordination* and *self-reconfiguration* of complex systems, or the so-called "system-of-systems" (SoS); and (2) the realization of using the methodology *to guide* system-of-systems design with a particular focus on *embedded* devices.

1.1 System-of-Systems

System-of-systems is a multidisciplinary area that involves system integration as a key to tackle complex tasks or problems, usually by means of composing multiple independently controlled systems together as part of a larger, more complex system [74, 60]. Listed below are examples of system-of-systems in different domains in which *autonomous self-reconfiguration* and *coordination* are highly desired features:

1. *Intelligent traffic*

   In an intelligent traffic system, a coordinator controls the traffic flow and schedules the occupation of the lanes to improve traffic efficiency [7]. Every vehicle is an individually controlled entity that can actively influence the overall performance and throughput of the traffic system.

2. *Platooning*

   A platoon system consists of a set of (semi)autonomous vehicles moving as a group without any mechanical connection while keeping an "optimal"

formation [36]. Each individual vehicle perceives the environment and acts on the basis of the perception and the overall control goals of the platoon. The vehicles must be capable to keep their desired positions in the platoon, in various platooning configuration modes (optimizing speed, safety, or energy consumption, etc.).

3. *Multi-robot system*

   The fourth industrial revolution has shown the trend of automation in smart manufacturing. How to design, configure, monitor and coordinate the individuals in a multi-robot system-of-systems to minimize human effort and reduce human interaction has become a hot and challenging topic [28].

4. *Smart home*

   Internet of Things (IoT) accelerated the development of the smart home industry. Smart home allows users to monitor home conditions such as temperature, humidity, luminosity, etc., and to manipulate home appliances remotely [120]. Microcontroller-based platforms have become popular and are playing important roles on dedicated tasks in home automation such as sensor data acquisition, being featured in low cost and low power consumption.

5. *Smart grid*

   Power distribution networks are increasingly developing towards the utilization of smart grids. A smart grid is a complex large-scale system-of-systems involving power plant, decentralized control, demand management, energy storage, and so on. The agents in an agent-based smart grid [27] are involved in the system coordination.

1.2 Cyber-Physical System

One class of system-of-systems integrating different *system types* are *cyber*-physical systems [138]. Computers and networks are organized to monitor and control one or more *physical processes* in such systems using feedforward and feedback loops, in which the physical processes and the computations are integrated with each other [75]. The concept of cyber-physical system was derived from the term *cybernetics* originally proposed by Norbert Wiener [130]

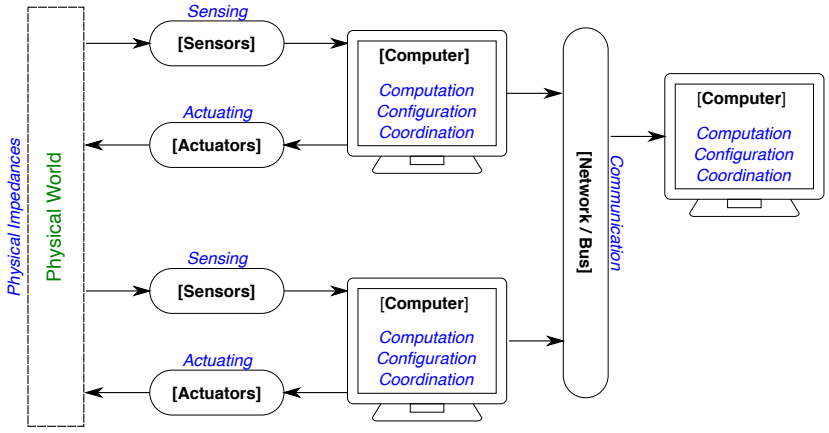

Figure 1.1: This figure demonstrates a general cyber-physical system. The bold terms represent the physical devices, while the italic terms are the primitives of a cyber-physical system, implying the behaviour and knowledge needed in the system at run time. The physical world is sensed by converting physical properties to electrical or electronic signals and is influenced by driving corresponding actuators. Multiple computers may be involved in a complex cyber-physical system for sophisticated algorithms or tasks.

in 1948. The major extra challenge with respect to pure digital systems-of-systems (such as cloud computing or on-line big data analytics) is that every physical part of the system has its own dynamics, which behave autonomously and whose behaviours are often even difficult to observe with sufficient detail or latency.

Nowadays, cyber-physical systems interact with the physical world using a digitized and discrete representation of the *physical impedances*, as depicted in Fig. 1.1. Physical impedances are energy generating, dissipating and transforming elements, in thermal, chemical, hydraulic, mechanical, pneumatic, and electrical domains. Any control of a cyber-physical system requires that these impedances are modelled and provided to the computers in advance, at the "appropriate" level of detail. In this thesis, most of the examples of the cyber-physical systems are in the robotics domain, in which the electronic and electro-mechanical impedances are dominant: many necessary parts of a robot, such as motors, springs, dampers, distance and gravity sensors, are monitored or driven by electronic signals.

Two major R&D focuses on cyber-physical systems in this decade are robotics including intelligent vehicles [98], and smart grid in power consumption

management [126]. As a result of the growing complexity of these cyber-physical systems, there is a trend towards an increasing demand of powerful computation capabilities and the needs of system-of-systems design. On the other hand, none of the existing systems and components have been designed to participate in a system-of-systems whose control requires *run time* reconfiguration of its sub-systems and child components. More in particular, their software interfaces are hiding too much of their internal structure and behaviour, preventing most opportunities for "overall optimality" when they are **part of** a system-of-systems.

A second major trend is the physical volume reduction of the computing devices stimulated by the development of the Internet of things (IoT). As featured in small scale, low cost and low energy consumption, the embedded computers nowadays are equipped with high performance processors due to the rapidly developing electronic technology. As service robots are being deployed in human-populated scenarios and sometimes constrained by available space, the physical size of robot controllers became a critical constraint in the human-robot interaction applications [37, 140]. In this context, embedded devices are considered as compact solutions to address the volume issue, while retaining sufficient computation power to handle desired tasks. In addition, embedded devices are usually equipped with various I/O ports and interfaces for communication and data acquisition, which is an excellent feature for integrating sensors and actuators in the system [111, 76].

Embedded devices such as ARM[1] processor based small-scale single-board computers, together with their derivative products like extension modules, capes [132, 131], reference books [87, 88] and websites, have occupied a significant part of the embedded computing and the educational product markets. In the meantime, programmable logic devices such as microcontrollers and field programmable gate arrays (FPGAs) are playing more and more important roles in industrial applications and have drawn significant attention in the world. The ARM-based, microcontroller-based and FPGA-based devices are the major platforms used in this thesis.

Modern cyber-physical systems may consist of multiple computing devices collaboratively working together for specific tasks, leading to a challenge on system-of-systems design. In this dissertation, we focus on the improvement of system-of-systems *design process*, by providing a *methodology* that covers *all cyber* aspects, to systematically compose, configure and coordinate all the computations and communications in and between sub-systems, on the basis of (more) formally specified desired behaviours of the overall cyber-physical system, and by exploiting the recent embedded technology and software developments.

[1] Advanced RISC (Reduced Instruction Set Computer) Machine

The focus of the work is **not** on the modelling or the improvement of the *physical* control algorithms.

As previously mentioned, this dissertation uses "running examples" of cyber-physical systems in the robotics domain, as robots present perfect examples of all connected primitives demonstrated in Fig. 1.1. The robotics domain is usually concerned with construction, manipulation and control of mechatronic systems that communicate and cooperate with other systems and the surrounding environment [83, 122], involving various types of sensors and actuators to interact with the physical world in different domains and applications [108, 61, 33].

The two trends and the research focus lead to the needs of composable and embeddable software design for robotic and cyber-physical systems, which invoke the following challenges:

1. **How to effectively represent the composition and decomposition for desired functions, capabilities and tasks in system-of-systems design process?**
2. **How to retain the flexibility of deploying a design on mixtures of embedded devices and "traditional" computers?**

1.3 Motivation

The above mentioned challenges induced the motivation of developing a composable and embeddable **system design methodology** to effectively and efficiently **use and reuse** existing hardware and software **resources**, and to support the creation of such systems from scratch. The term **system** in the context of this dissertation covers, but is not bound to, mechatronic and robotic systems; it refers to physical entities involving actuators, sensors as well as computing units and control logics to carry out tasks with **end-user specified quality of service** expectations. Within the context of the above-mentioned challenges, this text was particularly motivated by the following additional and more detailed challenges:

1. **How to model, identify and utilize the hardware resources in a systematic way to deploy the driving and computing software to optimize the design for robotic and cyber-physical systems?**

   A common characteristic shared by microcontroller, FPGA and single-board computer is the rich set of peripheral interfaces. This characteristic

is essential for communication, sensing and actuation needed by cyber-physical systems for intensive computation and interaction with the environment or human. The reality of various interfaces on different type of devices motivated us to explore the optimal use of device **resources**, and to guide the *configuration* and *coordination* of available computations for system-level **capabilities** that these devices should contribute to.

2. **How to effectively demonstrate, introduce and teach system-of-systems design and software programming, by motivating students using simple, compact, low cost and efficacious teaching materials?**

   During this thesis' research, it became apparent that providing a system-wide methodology would, in itself, have minor impact, because several trials in real classes made it clear that students have a very hard time to see the forest for the trees, for the simple reason that systems-of-systems have so many different parts, and so many potential capabilities and interactions. Hence, the decision was made to invest in making a start with education material (including both hardware and software) especially targeted towards the goal of illustrating the design methodology with simple and concrete examples. Along with the rapid development of simple microcontroller based computing platforms and single-board computers, electronic prototyping is getting more and more popular in engineering education. Moreover, the miniaturization of fabrication machines [45] has drawn a lot of public attention together with the rapid expansion of the fabrication laboratory, the FabLab [86]. It is nowadays much easier for designers to test their fresh ideas and bring them to life by the help of digital fabrication equipments, such as laser cutting machines and 3D printers. These developments were amply exploited in the above-mentioned educational challenge.

1.4 State of the Art

This section provides a survey of existing techniques, methods and tools in system design. The limitations of the state of the art with respect to satisfying the requirements of this research are concluded at the end of this section.

1.4.1 Levels of Abstraction: Capabilities of Systems

Levels of abstraction was originally defined by Dijkstra [32]. A conceptual framework was proposed in the discussion for reaching a logical design of a

system that could be conceived as a hierarchy of *levels*, in which the lowest levels are often the closest to hardware. One or multiple independent abstractions could be implemented on the same level, and each level is composed by a group of related functions. The concept of levels of abstraction has been considered effective in system design in general, and it is widely used in modern system design processes.

Liskov [79] divided the construction of a software system into three stages: design, implementation and testing. Limited by the computation power and programming tools in the 1970's, it was rather difficult to verify the design methodology, which aimed at facilitating the definition and realization of system modularizations. Nevertheless, seeing *the identification of useful abstractions* as a key to the design is still an important guideline to software programmers and system designers, especially in the robotics domain. For instance, in the popular Robot Operating System (ROS), a `rosbridge` provides an additional level of abstraction on top of ROS [22, 23] to integrate ROS and non-ROS functions.

The definition of levels of abstraction is intuitive; however, it is usually difficult to determine which (or what) levels should present in a system. Modern software frameworks in robotics domain such as Open RObot COntrol Software (OROCOS) [14] and ROS allow designers to choose the abstraction levels for given applications. Two popular abstraction levels in software are *tasks* and *functions* [133, 11]: tasks are problem solving oriented, while functions practically serve as tools to carry out specific tasks. Necessary knowledge concerning the *environment*, the mechanical, electrical and electronic *system properties* is required in software, so that a system is **capable** to complete desired tasks correctly and effectively.

The term **capability** is rather broad in the context of system design. In robotics domain, various capabilities such as colour detection [92], feature recognition [6] and collision avoidance [121] have been discussed and used in many research applications. Buehler [17] defined a *capability* as a simple functional element which can be part of many different tasks. This definition implied that capabilities are *reusable* functions for tasks. However, **how to create** capabilities in the system design process is not yet clearly specified. In this thesis, we **explicitly model the *"how-to"*** by introducing system specific parameters, and we also propose an effective approach to make use of the capabilities using **event loops**.

1.4.2 Separation of Structure and Behaviour

In the discussion of software reliability [79], Liskov presented a methodology to produce a program structure that facilitates the proof of software correctness

on complex systems. In the discussion, the term **complex** is defined two-fold: 1) many system states are presented in a system, and it is difficult to organize a logical program to handle all states correctly; and 2) building the system requires the coordination of individuals. The **complexity** of a system comes from two sources: complexity brought in by functions, and complexity in the connections between the modules.

The two sources mentioned above implicitly separated *structure* and *behaviour* when describing the software of a system. Efforts were spent on the explicit separation of structural (or architectural) and behavioural aspects in system design in order to increase the reusability of behavioural functions. For instance, *sense-plan-act* (SPA) was one of the first robot control methodologies to define a robot architectural pattern [90]. While the three entities of SPA, *sense* (acquiring sensor information), *plan* (computing actions and updates) and *act* (executing the planned actions), together with a unidirectional flow mechanism, are easy to understand by humans, and hence qualify as appropriate "abstractions". This particular *choice* is unfortunate, since the concepts are **intrinsically coupled** with each other via the physical world, and it is therefore **difficult to scale** any system design that is structured on these chosen abstractions.

Some programming and modelling languages support modelling structure and behaviour separately. The Unified Modelling Language (UML) [94] provides a set of diagrams for modelling different aspects of software to depict the structure graphically. Designers may use UML to diagram functional compositions together with the Architecture Analysis and Design Language (AADL) to define runtime behaviour [25].

One of the preferred hardware description language to program FPGA devices, VHDL[2], requires the designers to define **ports** in the individual *entity field* so that the behavioural processes implemented in the *architecture field* could exchange data with other entities through interconnected ports. Every entity in VHDL could be wrapped as an Intellectual Property (IP) core, which is highly reusable in different designs. Both Xilinx and Altera, the two biggest FPGA chip designers in the world, provide mature tool chains to facilitate the design process, from modelling to implementation of an FPGA-based system.

Some system design integrated development environments (IDEs) also support, either implicitly or explicitly, the separation of structure and behaviour in the design process. For example, A nice feature in Xilinx Vivado is the so called "partial reconfiguration", which allows the system to change its behaviour at run time without influencing the overall structure, for instance replacing an edge detection algorithm by a colour detection algorithm in an image processing application. Allowing *reconfiguration* at run time could address the resource

[2]VHSIC (Very High Speed Integrated Circuit) Hardware Description Language

exhaustion issue on those FPGA chips with limited flip-flops and memory. Although this feature could be used **only** with the latest Xilinx device families, the design model behind shows the trend of the separation of structure and behaviour.

Modern robotic systems are increasingly depending on run time use of knowledge, however, most of robotics projects with software engineering focus miss explicit structural models. Scioni et al. proposed a composable structural *meta model* and its domain specific language NPC4 [115]. The paper advocates to represent the **structural properties** of systems using *node*, *port*, *connector* and *container* as primitives, and *contains* and *connects* as relationships. NPC4 formalizes the structural properties of systems, and meanwhile it supports behavioural composition on top of the structural model. NPC4 practically guided the development of the system design meta model in this dissertation.

In the current existing modelling languages, for instance AADL, the representation of *containment* and *connectivity* are textual. It allows users to indicate the behaviour in a variable's name, such as `motion_sensor`. It is a significant advantage for human to understand, however, it also implies that the meaning of the variable is fixed, it is not possible to change them in the lifetime of the program, and no further formal information can be linked to the name to facilitate the (semi)automatic integration with other systems. In other words, as the containment and connectivity are expressed in text-based form, the design is bound to the compiler of the modelling language and therefore it is difficult to transfer the structural model from one implementation to another, which reduces the reusability of the model, especially at run time.

1.4.3 Formal Modelling in System Design

Model Based Engineering (MBE) or Model Driven Engineering (MDE) technologies provide a promising software development approach to improve software quality. The core of MDE is to define a modelling language, known as a **meta model** [3], which is suitable to cover the relevant aspects of a particular domain. The major benefit brought in by this approach is a **clear separation** of *domain knowledge* from *technical implementations*, and concrete models can be specified using the meta model. The analysis, validation and execution of the concrete models could be efficiently facilitated if the **formalized meta model** is presented [65].

A **formal model** of a system is a mathematical model at some specific levels of abstraction. The purpose of *formalizing* a model is to create a complete intermediate step between human and machine understanding of the model, which is in contrast with *conceptual model* that only helps people

with understanding a system. The necessity of using formal models in system design has been discussed in [20], [62], [50] and [52].

Petri net [100] is a mathematical model introduced by Petri in 1962 to describe distributed systems. Petri nets consist of places, transitions and arcs. A transition and a place is connected by a directed arc. Tokens residing on places represent the state of a Petri net. Hrúz and Zhou discussed Petri net properties in [55].

A hybrid automaton is a formal mathematical model for a mixed discrete-continuous system [54], it exhibits two types of state changes: discrete transitions occur instantaneously and continuous transitions occur along with time [106].

Xilinx Vivado Design Suite [56] offers formal modelling tools for FPGA-based system design. It allows designers to use the *Tool command language* (Tcl) [97] integrated in the Vivado environment as scripting language to formulate a design, as well as to interact with the design environment. The formulated formal model of the design is shown graphically in the design tool. It is also possible to use Tcl to realize the previously mentioned *partial reconfiguration* (which separates the modelling of structure from behaviour) in Section 1.4.2. Simulink offers a similar and formal way to build models using MATLAB commands. However, these formal models are determined at compile time, i.e., once a formal model is created, the structure of the model would be fixed *at run time*, and therefore isolated from run time recomposition.

1.4.4 System Synthesis

This section discusses the state-of-the-art of system synthesis from two particular aspects: software architecture and the modelling languages and tools for system behaviour composition.

Software Architecture

A software architecture is defined in [4] as: *The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.* In this book, Bass et al. implied that an architecture is a set of software structures as well as an abstraction of a system that selects certain details and suppresses others. It has been pointed out by Kortenkamp and Simmons [71] that designing a robot architecture is an art due to the trade-off among different requirements in the system such as usability and flexibility. A well-conceived architecture together with programming tools that support the architecture often help to manage

the system *complexity*. A framework supporting software architecture-based development of systems is accessible to non-experts in robotics and facilitates adaptation in complex and dynamic environments [85]. Essential aspects of robot system architectures involve the underlying paradigm, the programming system as well as the communication mechanisms [84, 9, 116].

Composition of Behaviour

Klotzbücher et al. [66] introduced an architecture-independent *coordination approach* to facilitate the composition of discrete behaviour of robot system. Coordination is essential for complex systems to compose and regulate discrete behaviours using state machine as event processors to determine how sub-systems or computational components should collaborate.

The Behaviour Interaction Priority (BIP) framework [5] allows creating complex systems by coordinating the behaviour of a set of atomic components. In the BIP framework, the coordination is based on two primitives (on top of the third one, *behaviours*): 1) *connectors* that specify possible interaction patterns between components; and 2) *priorities* that select possible interactions, which imply the *order* of interactions as scheduling policies. BIP layers a system using the above mentioned three primitives with a clear separation between behaviour and structure. The composition of components is realized by parameterized binary composition operators.

The major difference between both approaches is that BIP implicitly mixes computation and coordination, and Klotzbücher et al. [66] explicitly separate them. The consequence is that the latter is somewhat more difficult to use, and the former is easier to prove behavioural properties *if* the mentioned coupling can be realised without any cost in time or other resources. However, this assumption does not scale at all with growing complexity in functionality and in connectivity, which is the main driver behind Klotzbücher et al. [66].

SysML [95] models the interactions with the physical world, and allows some basic verification of properties on the models [77, 70]. It is a customized version of UML2 fitting the system engineering domain with extensions such as control and data. In this thesis, *SysML flowport notation* is used in the coordination tool modelling (state machines) to describe the transition between two states. SysML is the result of a unification effort, which led to such a huge "standard" with so many "semantic variation points", that interoperability is limited, and support is only available via non-embeddable IDE tools.

The Ptolemy framework [35] supports creating systems using heterogeneous components. Different paradigms are allowed to be used in the composition of

components in the Ptolemy framework, these heterogeneous components are composed using a model of computation that ensures control and dataflow among the components. A composition can be aggregated with other components on a higher level with a different model of computation. Ptolemy classic discrete event model of computation uses the firing rule, in which a block is active when new data is presented on at least one input. In this way, the schedulers and the blocks are coupled with each other and therefore limited the flexibility when rescheduling is needed at run time.

LINX [57] is an open source project supported by Digilent to facilitate embedded application design using LabVIEW. It turns embedded devices such as Beaglebone Black, Raspberry Pi and Arduino Mega2560 to I/O hubs for the Virtual Instrument (VI) program running on a PC [113]. The embedded devices must be programmed with specific firmware running on the on-board processors, and they are actually *tethered* with the host PC where the LabVIEW program is running. The on-board processors are in charge of handling data communication instead of computations: the computation tasks are taken care by the LabVIEW program in this case.

Aledyne Engineering and TSXperts launched a commercial compiler *Arduino Compatible Compiler for LabVIEW* (ACCL) [123] in 2015 that compiles LabVIEW VI programs for Arduino devices, so that these devices can run independently after programmed by LabVIEW. The ACCL compiler filled the gap between graphical programming language and popular low-cost embedded devices, however, as most of the resources such as libraries for extension shields available in the communities and forums are in the form of code (C/C++, Python etc.), it is still a challenge for the ACCL compiler to enrich the support of using existing code. Another drawback of the ACCL compiler is, it supports only several Arduino boards. These drawbacks practically reduced the motivation of users to switch to ACCL, especially when different types of embedded boards are involved in a project.

Simulink allows users to develop, simulate and deploy algorithms that run standalone on embedded devices such as Arduino Uno and Mega2560. A broad family of Arduino devices as well as Ethernet extension shields are supported. Comparing with LINX and ACCL for LabVIEW, Simulink is more flexible on simulating and deploying a design, and it also supports code generation for embedded devices. However, the available blocks and models for embedded devices like Arduino in Simulink are rather basic, users still need to implement their own drivers for sensors such as an accelerometer. Meanwhile, similar with the ACCL, the current existing libraries can not be used directly. Moreover, although the code generated for the embedded devices are neat and highly

readable, a significant overhead is often involved[3], which could bring in memory consumption issue in small systems and therefore preventing developers to use those devices efficiently with limited memory.

Modelica [40] is a language for modelling the *behaviour* of complex cyber-physical systems. It provides libraries of standard physical models and compile time computation to create large models. The model behavior is based on ordinary and differential algebraic equation (OAE and DAE) systems combined with discrete events. The integration of Modelica and UML/SysML in *Modelica graphical Modelling Language* (ModelicaML) enables to create and maintain Modelica models [110]. ModelicaML is designed towards the generation of Modelica code from graphical models.

1.4.5 Conclusion of Literature Survey

Most of the approaches mentioned above mainly target use by human developers only, and not the run time processing by controllers in cyber-physical systems. The ambition of supporting *autonomous coordination*, *self-reconfiguration* and *run time recomposition* is a key long-term focus of this research, and it requires the system design methodology to support **explicit separation** of structure and behaviour, and it should have a **formalized model** when composing the structure of behaviours. Most modern modelling languages and tools focus on the abstraction and extensions of system behaviours with rich sets of primitives, but attach less importance to the embeddability and composability at run time. Moreover, most of the related works do **not** explicitly separate coordination and computation, which is in practice essential for system-of-systems to regulate the computation behaviours by determining how the sub-systems effectively collaborate with each other. This thesis explores the best practices in system design towards the above mentioned ambition by pursuing several research objectives set in Section 1.5, using the approaches listed in Section 1.6.

1.5 Research Objectives

The following research objectives aim at answering the questions posed in the motivation and overcoming the weaknesses in the state of the art.

[3]The HEX file generated by Simulink for turning on an LED on Arduino Uno requires 3622 bytes of program memory and 234 bytes of data memory with RAM efficiency configuration, while the same behaviour can be realized with 846 bytes and 9 bytes respectively by hand coding.

1. **Methodology: Developing a systematic approach to model cyber-physical systems, and to guide the creation of reusable, flexible and adaptable software for robotic and cyber-physical systems, as well as system-of-systems.**

   Developing a **meta model** to facilitate modelling and describing cyber-physical systems and system-of-systems is the primary task of this dissertation in this particular objective. First of all, this meta model should cover hardware *resources* mentioned in the first question in Section 1.3; next, it should allow people to choose the most appropriate hardware devices in system design as an answer to the second question.

2. **Software framework: Developing and implementing a composable software framework that is consistent with the approach in the first objective.**

   This objective was triggered by the needs of effective and efficient software design, considering reusability and composability as more important targets than usability and simplicity. Hence, the target group of this research are the professional system-of-systems developers, and not their end users; "ease of use" must be added later, and by others, via user-centric *software tooling* around the models and software provided in this research. As the goal is to support composition of complex models by combining simple ones while maintaining stability and robustness of the overall system behaviour, **composability** is of primary concern. Software engineers would never prefer to write code from scratch as the cost could be quite high. Instead, reusing existing code and libraries is more reasonable to reduce the overall labor cost. Under this premise, *composable modelling* is the preferred road towards enhancing software *reusability*.

3. **Software compatibility: Maximizing the compatibility of the software framework for different embedded device families.**

   The software framework for cyber-physical system design is not competing with existing robot control software such as ROS [103] and OROCOS [13]; instead, it can be integrated in these mature software frameworks, in the form of ROS packages or OROCOS components. In addition, the current versions of ROS and OROCOS only support operating system based embedded devices; while this thesis endeavors to provide a complementary but small software framework that is applicable on both OS-based and bare metal embedded systems. The term *embeddable* does **not** imply the dedication to embedded applications, instead, this software framework is applicable on both embedded and traditional computers, being aware of resources, capabilities, tasks and the cyber-physical world [15].

The software framework is expected to be *deployable* on the devices from different manufacturers and families, this requirement makes C language the primary choice for the implementation as it is supported by most of the microcontrollers and processors.

4. **Best practices: Find the best practices on integrating sensoring, actuating and computing tasks on heterogeneous devices, and on utilizing the available hardware resources, in the context of the systematic approach.**

   This objective involves the exploration of various types of computation devices and their peripheral interfaces, including practical experience on system design with FPGA. Due to the modern richness in computing platforms, this objective is the most important contribution in the challenge of helping students and developers to see the proverbial forest for the trees. The best practices contributed by this thesis use the form of the well-known *software patterns* concept, in that they provide motivated solutions to often occurring design problems, including documentation of where and why these solutions have already been applied in practice.

5. **Education: Using the meta model developed for cyber-physical system design and the software framework, together with the best practices obtained to build robots and setups to demonstrate, introduce and teach system-of-systems design, and to facilitate the educational activities.**

   This objective is essential for verifying the validity and feasibility of the system design approach, meta model and software framework proposed in the previous objectives. During the duration of the thesis, these educational efforts have been used in real-world courses in embedded system design and robotics, at KU Leuven and TU Eindhoven.

1.6 Approach

The research of this dissertation relies on the five approaches below.

1.6.1 System Design Phases

A system design process often starts with an inspiration or a simple concept, and eventually ends up with a concrete entity that carries out the required functionalities. In this dissertation, we define four phases and advocate to use *design patterns* in the system design.

- **Design Pattern**

  A design pattern is a general repeatable solution to a commonly occurring problems in system design [117]. A pattern is not a finished design but a description or instruction of how to solve similar problems. Design patterns can be used through all the design phases, they introduce extra constraints and bring in trade-offs to help formulating the design in a structured way.

  An important design pattern in software programming is the *mediator pattern*, which promotes loose coupling by keeping objects from referring to each other explicitly, so that their interaction can be independently varied [42]. The mediator pattern defines an object that encapsulates how a set of objects interact with each other. It is a useful solution to tackle mismatches on communication protocols [78], and it supports coordination of multiple hardware devices to achieve a system level behaviour as hardware and software interfaces [34]. In this dissertation, the development of the meta models actually reflects the thinking of using mediators to bridge the functionally or semantically mismatching parts in system design. In addition, as one of the major contributions, we also demonstrate how to recompose the decoupled parts, as a systematic methodology of *composition*.

1. **Concept**

   A concept is a preliminary idea that reflects the intention and direction to solve problems.

2. **Model**

   The original idea could be formulated as a conceptual (meta) model with primitives. In this dissertation, typical conceptual models of systems are depicted in Fig. 2.3, Fig. 2.9, Fig. 5.3, Fig. 5.8, Fig. 6.3 and Fig. 6.4.

   A conceptual model is formalized as a formal (meta) model, i.e., a systematic approach to solving similar problems. (Meta) models are expected to be reusable and composable at run time for system flexibility and performance. A well-structured (meta) model benefits from design patterns in the modelling process.

3. **Implementation**

   Implementation is the realization of the formalized (meta) model to carry out the desired functions, capabilities or tasks using specific design patterns. The implementation usually requires an encoded, computer readable formal (meta) model.

4. **Deployment**

   The implemented software code is eventually deployed on suitable platforms or hardware devices to comply the proposed concept in practice.

1.6.2 Cyber-Physical Stack for System Integration

We suggest a concrete set of *levels of abstraction* to describe and design cyber-physical systems, using a **Cyber-Physical Stack** (CPS) meta model.

As presented in Fig. 1.1, we start with the electronic **devices** as the lowest **layer** with specific hardware details, then move toward the software layers of **functions**, **capabilities** and **tasks**. The CPS meta model is formalized and encoded, and eventually implemented as a software framework.

1.6.3 (Meta) Model Formalization

A *conceptual model* needs to be formalized as a *formal model* so that it can be interpreted and understood by computers. The inspiration and motivation stemmed from the ontology theory in engineering [10]. The term *ontology* has been adopted by computer scientists to denote structured framework to represent information and knowledge [89], it refers to an explicit specification of a conceptualization [46]. In [89], the authors defined *formal specification* and *informal specification* to distinguish the two levels on ontology, in which the formal specification constitutes an implementation of the ontology in machine-readable form, while the informal specification expresses the definitions of the formal specification in human-readable form. To achieve the ambition of autonomous self-configuration, the developed Cyber-Physical Stack meta model must be *formalized* for the computers.

We use several existing meta models to formalize the Cyber-Physical Stack meta model. One of these models is the Composition Pattern [127] which extends the separation of concerns [104] in robotics. The Composition Pattern was proposed for creating composite tasks by assigning functional roles to each component of the system. The task dependency graphs in [114] further formalized the task composition. While the above meta models represent behaviours, the NPC4 meta model [115] models the structure of the systems. The Cyber-Physical Stack conforms to [8] the above mentioned meta models and it formalizes cyber-physical system models for composable and embeddable applications.

There are three key elements involved in (meta) model formalization: *primitives*, *relationships* and *constraints* [115]. For example, in the general cyber-physical

system demonstrated in Fig. 1.1, the physical impedances are modelled as the medium that connect the physical world and the computations. A formalized model of the physical impedances on energy structure is the *bond graphs* [12], in which the primitives are energy storage, transformation and exchange; the relationships are formalized by the differential equations; and the constraints are brought in by the topology, i.e. containment and connectivity of the bond graphs that affects the physical behaviour. The primitives of a formalized *smart grid* model may include computing devices, power stages, controls and automation; the relationships are reflected by the network connections; and the constraints are brought in by the rules of electricity distribution.

1.6.4 Life-Cycle State Machine for Coordination

Coordination is essential on animating a cyber-physical system with desired vitality on computation. In this dissertation, We use a *Life-Cycle State Machine* (LCSM) to facilitate the *coordination* and the *configuration* of activities and behaviours. As physical processes are controlled by *computations* running on cyber-physical systems, the computation behaviours are expected to be well coordinated to meet the functional requirements.

1.6.5 Experimental Setups for Education

FabLab is a small-scaled workshop offering creative space for designers to test out ideas and to bring them to life. By utilizing the digital fabrication equipments, we are capable to build low cost gadgets, setups or simple cyber-physical systems to demonstrate the usage of the Cyber-Physical Stack in educational activities.

1.7 Contributions

The main contributions of this dissertation are listed below.

1. The Cyber-Physical Stack meta model

The Cyber-Physical Stack (CPS) meta model is proposed on the basis of a hierarchical design method derived from [115], using containment and connectivity as the structural description of a system. The main contribution is in defining and explaining a novel *array-of-integer* (AOI) representation to formally describe the structure of computation behaviour by computer readable arrays, as well as in specifying *capabilities* as a knowledge-dependent primitive

in the CPS to strengthen the composability and reusability. The Composition Pattern [127] is consolidated in the CPS context to assist behaviour composition: the concerns are primitives in an *event loop meta model*, which facilitates the behavioural modelling and improves the system scalability. The Cyber-Physical Stack meta modelling is discussed in Chapter 2.

2. The Life-Cycle State Machine

The Life-Cycle State Machine (LCSM) is an essential coordination approach to obtain desired behaviour by properly configuring the computations when required. As the fourth iteration of the coordination technique on the basis of the result of Soetens [119], Klotzbücher et al. [68] and Vanthienen et al. [127], we explicitly abstract **resource** and **capability** as the two properties in the life-cycle of a system or computation, as the behaviour of a capability is determined by the contributing resources. The LCSM is realized by means of a **composable Finite State Machine (cFSM)** meta model derived from the concept of rFSM presented in [68] with composable and embeddable features. The cFSM meta modelling and the LCSM are introduced in Chapter 3.

3. Composable and embeddable software framework

Aiming at assisting the software design for robotic and cyber-physical system, a composable and embeddable software framework is developed to implement the CPS and the cFSM meta models. The code is written in C language without bothering any platform dependent library, it can be used on any hardware platform as long as C language is supported. The CPS and the cFSM core engine currently amounts in total 1850 lines of code[4], which is compact enough for embedded applications. The implementation details and thinking are discussed in Chapter 4.

4. Best practices of using embedded hardware devices in system-of-systems design

Various embedded hardware platforms that can be utilized as device resources are used and discussed in this dissertation. Besides the microcontroller and the general purpose processing units, specific programmable logic devices, the FPGA and FPGA with a System-on-Chip (FPGA SoC) are exploited to expand the resources on the devices layer of the Cyber-Physical Stack meta model. The best practices of using embedded devices in cyber-physical system design are shared in Chapter 5.

5. Low cost setups for educational activities

Several low cost educational setups were created using the digital fabrication

[4]Calculated using the *cloc* tool in Linux

equipments. These setups are the physical platforms on which the CPS and cFSM software are deployed and tested, they have been used in the *Embedded Control System* lectures, student projects, PhD schools and seminars. A step-by-step example of system design is demonstrated in Chapter 6.

Chapter 2

Cyber-Physical Stack

This chapter formulates the answer to the questions posed in Chapter 1 by pursuing research objective 1: *Methodology: the Cyber-Physical Stack meta model*. We follow the system design phases introduced in Section 1.6.1 to develop the meta model and illustrate how to use it by a simple, compact but complete example to design the *computation structure* a cyber-physical system; followed by an event loop meta model to improve the scalability of the *computation behaviour*.

2.1 Design Patterns

We exploited, improved and created the following design patterns in the modelling process of the Cyber-Physical Stack (CPS) meta model:

- The Composition Pattern
- The Coordination-Configuration Pattern
- The Computation Behaviour Composition Pattern
- The Event Processing Pattern for Cyber-Physical System Operation

2.1.1 The Composition Pattern

In this dissertation, the *concerns* in the Composition Pattern [127] are considered as key elements in cyber-physical system design process as they determine the

overall system behaviour. In addition to the existing concerns, we advocate *logging* as an additional concern in the Composition Pattern to collect, record and present data at run time for human interaction and system diagnosis.

In the software design context, separation of concerns refers to the thinking of dividing a system into reusable and relatively independent parts originally discussed in [104]. It considers communication, computation, configuration and coordination as *concerns* with minimum overlapping. The 5Cs principle [16] introduced *composition* as the fifth concern, afterwards it is extended with *scheduling* and *monitoring* as two additional concerns, and proposed these concerns in a *Composition Pattern*. As a major contribution, this dissertation extended the Composition Pattern from task level to system level to assist the system-of-systems design.

Review of the Concerns for future advanced system-of-systems

In the context of this thesis, functions are expected to be running on target devices. A *targetable* device is a piece of hardware that allows designers to deploy code segments and to execute them to carry out specific **computations**. A non-targetable device, on the contrary, can not be programmed, yet they could be **configurable** (configuration). For instance, in the platooning example in Section 1.1, the relative positions of vehicles can be configured to change the platoon formation.

A complex computation or system-of-systems is **composed** (composition) by multiple child computations or sub-systems. The operation of the sub-systems, or the execution of the child computations must be **scheduled** (scheduling) with a specific sequence to obtain the desired overall performance or behaviour, especially when the resources shared in the system are limited or constrained. For instance, in the intelligent traffic system example, lane occupation must be scheduled in order to maximize the traffic efficiency and throughput.

Coordination refers to the process of managing and monitoring functional computations for system intended behaviours [67]. The coordination process is usually initiated by an event, resulting in the reconfiguration or rescheduling of computations in the system.

Communication is involved when inter-device messaging is required. In the smart home system, a microcontroller may acquire the room temperature from the temperature sensor and then sends it to the central control device through dedicated communication channel. The vehicles in a platoon also need to communicate with the central controller or coordinator to report their current status or to receive commands.

Monitoring is required to observe system performance by verifying data and events. It is a crucial concern in a system as proper reactions are needed to regulate or change the behaviours. For example, as a sub-system, a power plant in a smart grid needs to monitor the energy consumption, generation and storage to optimize the productivity and efficiency.

Logging is an optional and complementary concern that allows designers to log data or to display them at run time for debugging, behaviour analysis and performance tracing purposes. The room temperature and humidity could be logged in files on the central computer in a smart home system for a daily overview of the room state, they could also be displayed on a mobile phone application at run time.

2.1.2 The Coordination-Configuration Pattern

The Coordination-Configuration Pattern was proposed by Klotzbücher et al. in the discussion of pure coordination [66], which advocates to split coordination and configuration to improve the reusability of coordination models. The Coordination-Configuration pattern is used in the Cyber-Physical Stack, we will discuss the details of this pattern and the improved coordinator model, the Life-Cycle State Machine in Chapter 3.

2.1.3 The Computation Behaviour Composition Pattern

This pattern contributes to the composability of the computation behaviour, and it guided the formalization of the algorithmic computation aspect of the CPS meta model. The behaviour of a system is influenced by the structural composition, the functions and the schedule. As most of the algorithmic computation models and modelling methods [102, 39, 109] as well as computer aided graphical modelling tools (such as LabView and Simulink) do not explicitly support run time recomposition, the Computation Behaviour Composition Pattern advocates composing a computation behaviour by specifying the structural containment, connectivity of functions and data, plus the execution orders or schedules of the computations so that run time recomposition is well supported.

2.1.4 The Event Processing Pattern

Event Processing (EP) is considered an increasingly *mainstream* view in information technology and it has been widely used in event-based control

in different system design solutions [99, 80]. The event processors capture events at run time and execute specific logic as response, and raise new events if required [107]. In this dissertation, the Event Processing Pattern contributes to the development of the *event loop meta model* which helps to improve the scalability of the proposed system design approach when integrating sub-systems in system-of-systems.

2.2 Capabilities

The term **capability** has been used in different context in many research. Fua and Ge [41] defined capabilities as subtasks in multi-robot cooperation. They suggested to decompose primitive tasks into smaller subtasks, which possess to be eligible for a task. Buehler [17] defines a capability as a simple functional element that can contribute to many different tasks in heterogeneous multi-robot systems, and the capabilities have dependencies on hardware devices. Buehler also advocated that a capability should abstract from underlying architectures at a medium level of granularity.

The definition of capability by Fua and Ge implies the *relationship* between tasks and capabilities, however, it does not decouple capabilities (subtasks) from tasks and therefore consequently weaken their reusability. Furthermore, the above-mentioned definitions did not explicitly specify: 1) *how* to compose capabilities in a well-structured way while keeping the reusability and composability, and 2) *how* to make use of the capabilities so that they fit in different systems of different scale.

In this chapter, we define "what is a system capable of doing" as a *system dependent* **capability**, in which additional *knowledge* is required when composing desired computation behaviour of a system. We formulate the answers to the above questions by introducing the *Cyber-Physical Stack* to handle the structural composition of computation behaviour (Section 2.3 to Section 2.5), using the design patterns discussed in Section 2.1, followed by an *event loop meta model* (Section 2.7) in the software architecture context to improve the *scalability* of the overall approach.

2.3 Conceptual Model of Cyber-Physical Stack

In this dissertation, we propose a four-layer structure to identify cyber-physical systems using **devices**, **functions**, **capabilities** and **tasks** as shown in Fig. 2.1.

The motivation of decoupling the four aspects in different layers in a system stemmed from the following facts:

1. Mature computing, sensing and actuating **devices** are widely used in many cyber-physical systems. Hardware engineers have been dedicating on electronic design to improve the performance of the devices along with the evolution of silicon technology.
2. Software **functions** including device drivers and algorithms are contributed by software engineers. These functions build up the overall functionality of cyber-physical systems and system-of-systems, running on top of the hardware devices.
3. Application builders attend to customer expectations, formulating them as **tasks** for robots and other cyber-physical systems, and they expect to collaborate with component builders to see these tasks realized by the above-mentioned functionalities.
4. System developers focus on configuration and coordination of these functionalities embedded in software architectures, to provide system **capabilities**. They want to optimize (i) the *usability* of their efforts to support as many application tasks as possible by simple recomposition, and (ii) the *reusability* of the available functionalities by stimulating the component builders to provide their artefacts in a form that support the required configuration and coordination flexibility.

There is a strong connection between functions and devices. As electronic devices are the physical carriers of computations, they must be initialized, configured and triggered by corresponding software drivers or driving functions. However, it is a *bad practice* to compose **tasks** directly with software **functions** in cyber-physical system design as it leads to loss of reusability of computation functions. On one hand, the software functions, especially device drivers are intimately linked with the device specifications: the software functions are more aware of *what* to do (or to drive) rather than *when*; on the other hand, it is essential for a robot to know *how* to complete a task, without being bothered by the hardware details. In this context, **capabilities** are needed as an *interconnection* between tasks and functions, as they know *when* to invoke the functions with reasonable purposes. Meanwhile, a task is aware of *how* the capabilities should be utilized for desired performance. Inserting the capabilities layer enhances the usability and reusability of the functions, and it helps to decouple tasks using existing functions as knowledge [64].

We consider **capabilities** as *resources* required by a robot or cyber-physical system to *execute* tasks, and as such *active* resources (that is, processes) they

form their own *level of abstraction* in the whole CPS context. Capabilities are *system* dependent, in that *system-specific knowledge* (such as mechanical, electrical and electronic properties of the target system) must be provided as **configuration parameters** in the capability model. Similarly, when using existing capabilities to carry out tasks, *environment-specific and task-specific knowledge* such as world models, object locations and task related constraints [72] must be provided at run time to correctly *configure* the executing capabilities.

We progressively introduce the primitives of the CPS concept by three figures. Fig. 2.1 demonstrates the concept of the four layers that compose the Cyber-Physical Stack, it is extended by introducing the behavioural concerns in the Composition Pattern as shown in Fig. 2.2. The connectivity is finally introduced in Fig. 2.3.

Primitives, Relationships and Constraints

We summarize the primitives of the *conceptual CPS meta model* in two categories: structural primitives and behavioural primitives.

1. **Structural**: block, port, connection, containment, connectivity
2. **Behavioural**: computation, communication, coordination, configuration, scheduling, monitoring and logging

Blocks are containers of functions or data, or, a set of child blocks. A block containing child blocks is a *composite block*. A *composite block* has the same semantic of *block* which can be contained as a *child block* by another composite block. Capabilities and tasks are composite blocks as they are composed by blocks containing function and data blocks.

As shown in Fig. 2.1, we use coloured **solid squares** to represent (composite) *blocks* on each layer: green for devices; red for functions; blue for capabilities; and orange for tasks. The *composition* of blocks is represented by the **solid contours**. Blocks that are contained in a contour on one layer form up a composite block for the upper layer, indicated by the**dashed lines** with the same colour code.

For instance, on the devices layer (L0), every green block stands for a piece of hardware device; several devices may be driven by a single function, represented by a red block on the functions layer (L1). The green blocks or devices are contained by solid red contours, each of which contributes to a function, as indicated by the dashed red lines. It is worth noting that in the Cyber-Physical

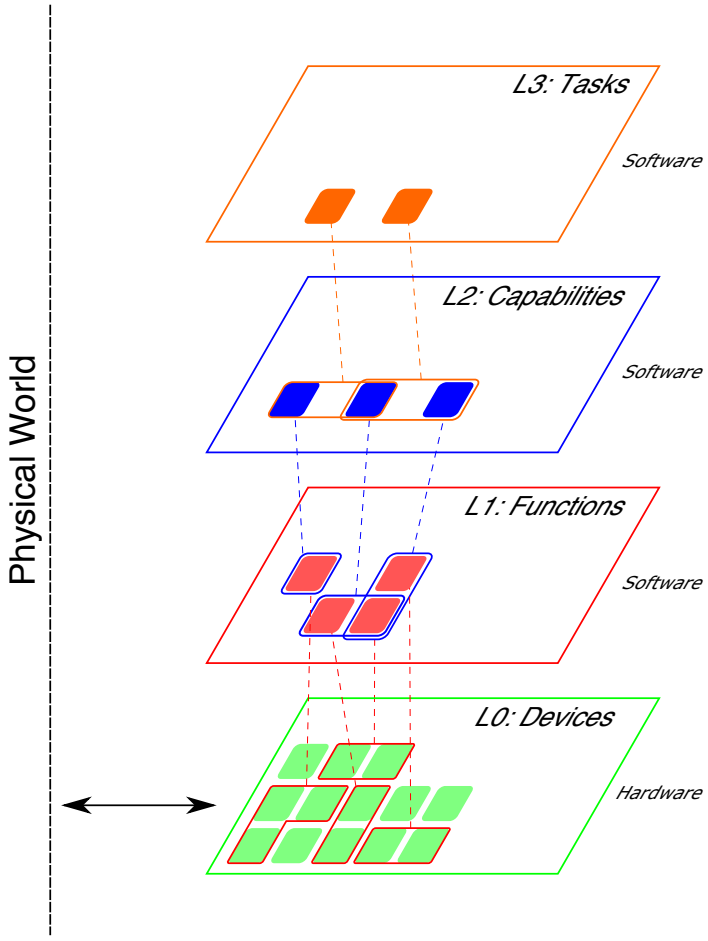

Figure 2.1: The cyber-physical stack contains 4 layers: devices (L0), functions (L1), capabilities (L2) and tasks (L3). L0 is the physical layer that contains devices to communicate with the physical world.

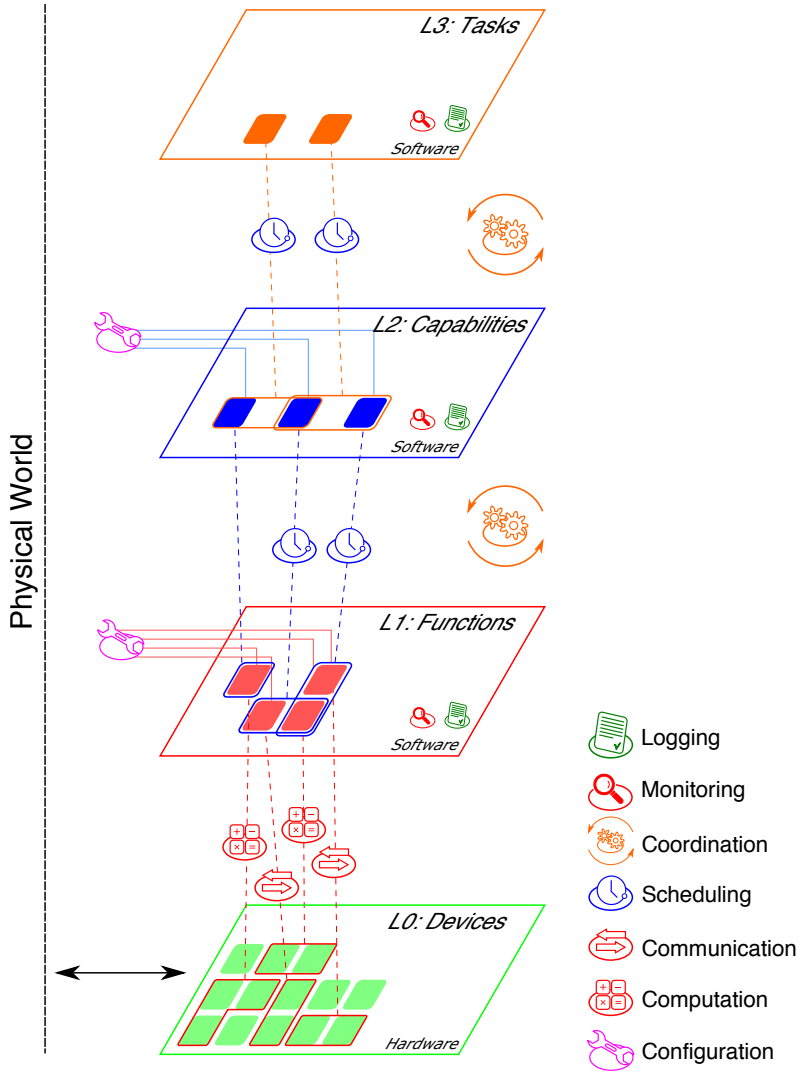

Figure 2.2: The concerns of the Composition Pattern are allocated in the Cyber-Physical Stack to emphasize the behaviours, indicated by icons for simplicity.

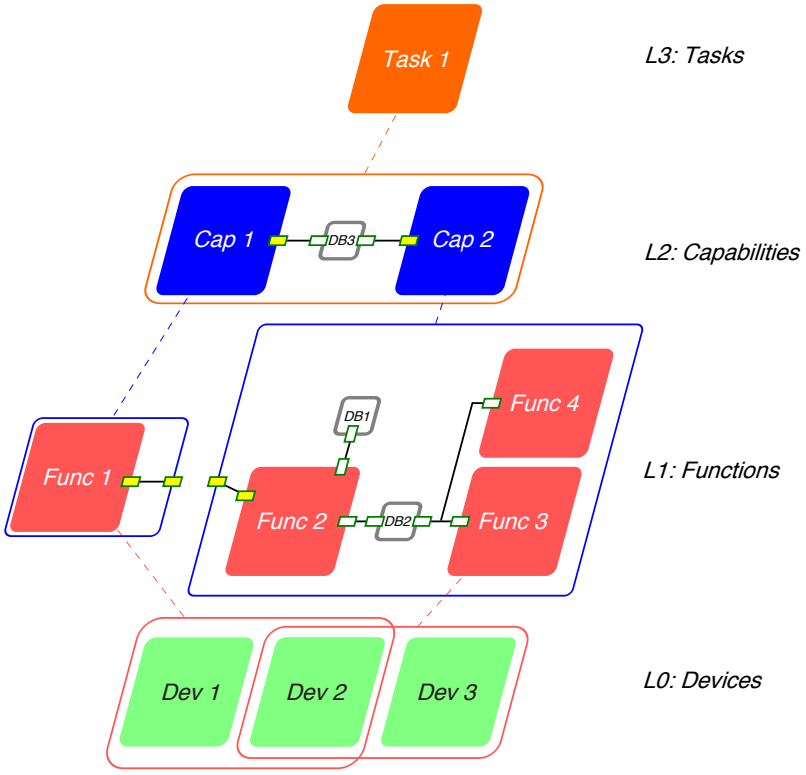

Figure 2.3: The connectivity involves ports and connections. A connection is a path to exchange data between two ports attached on blocks. Internal ports are mapped with external ports in composite blocks.

Stack, not all the devices are necessarily used by functions, as implied by the dodged green squares on the devices layer (L0). Meanwhile, as CPU and memory are needed by almost every function, they will not be explicitly placed on the devices layer (L0) for simplicity.

Fig. 2.2 introduces the behavioural concerns brought in by the Composition Pattern. *Communication* functions and algorithmic or device driving *computation* functions on L1 are physically deployed on the devices including sensors, actuators and computing hardware on L0, they can be configured by *configurators* for desired behaviours at run time.

Scheduling is needed by composite blocks (containing two or more child blocks) to specify the execution sequence of its child blocks.

Coordination is required between adjacent software layers to regulate the block behaviours. Coordinators trigger configurators to reconfigure or reschedule the computations in run time.

Events *monitoring* and data *logging* are applicable on all the layers, multiple monitors and loggers may exist in the same system.

In practice, every behavioural primitive can be realized as a function on L1. However, it is not obliged to include all the behavioural concerns in a system, nor to explicitly create functions for all the concerns. For example, a logging device such as an LCD can be accessed by different function blocks. Actually, in many embedded systems, some of these concerns are not presented due to the dedicated and repetitive behaviours.

Fig. 2.3 presents the details of *connectivity*. We use grey-edged square to represent a data block in the figure. Both data blocks and function blocks can be attached with *ports*, as represented by the tiny rectangles on the edges of the blocks. A *connection* is a path for data exchange involving two *ports* as shown in Fig. 2.3.

In the figure, each of the two capabilities `Cap 1` and `Cap 2` has one port, they exchange data with data block `D3`. `Cap 1` is composed by `Func 1`, and `Cap 2` is composed by `Func 2`, `Func 3` and `Func 4`. `Func 1` and `Func 3` are functions involve hardware devices, while `Func 2` and `Func 4` are algorithms running on CPU.

Function `Func 2` has three ports, the ones attached on the top and on the right edges are connected with data blocks `D1` and `D2`. The port on the left edge, indicated with yellow fill, is mapped with the port on `Cap 2` so that `Func 2` is the one actually connected with data block `D3`. The same indication is rendered on the ports of `Func 1` and `Cap 1`.

The relationships of the CPS concept are described by the containment and the connectivity between the blocks. Each layer has particular constraints that could influence the blocks on the same layer or on the other layers. The devices are constrained by the physical properties of hardware, i.e., CPUs, RAMs, buses, IOs, peripherals and so on, which bring in a lot of constraints to the algorithms and computations. For instance, on a microcontroller, the number of pins limits the amount of attached sensors; and the frequency of the processor limits the computation performance. The capabilities are constrained by the available functions, by the tasks they contribute to, and by the physical impedances. For example, the joint limits of a robot arm determine the range of the end effector, and therefore influence the capability of the robot; the tasks are constrained by the available capabilities and task related constraints such as control laws, obstacles and world models.

2.4 Formalizing the CPS Meta Model

In order to support run time recomposition and reconfiguration of the computation behaviour, we need to formalize the structure of computations in the CPS meta model. The formalized meta model is ready to be implemented and deployed on physical hardware. It is part of the whole CPS meta model, but is already mature for the embedded focus, and has been tested in the teaching activities of the Embedded Control System and student projects.

The formalization is directed by the Computation Behaviour Composition Pattern introduced in Section 2.1.3. This pattern focuses on the structural composition of behaviours, using *block, port, connection* as primitives (in accordance with the NPC4 [115] meta model), and *containment and connectivity* as relationships. We propose a novel array-of-integer (AOI) representation to describe the containment and the connectivity in the structural composition, using non-negative integer arrays in accordance with the pattern, and introduce scheduling list for the execution sequence of functions to compose the behaviour. Several rules are advocated as constraints in the formalization in order to ensure the correctness of the arrays.

In the formal model, the semantic of *block* is the same as that proposed in the conceptual model. Any block can be a composite block container, despite of the block types. Strictly speaking, **scheduling**, as one of the concerns, can be implemented as a function and be put in an F-block. However, as it is indispensable in every composite block, it can be standardized as a *scheduler block*. The separation is not violating the conceptual model, and is still consistent with the advocation of the Computation Behaviour Composition Pattern.

Consideration of Embeddability

The formulation of the AOI representation for containment, connectivity and scheduling arrays is carried out with the consideration of *embedded-friendliness*.

On embedded devices, especially microcontroller based platforms, program memory and data memory are sometimes limited to several kilobytes. This limitation triggered us to be cautious on the use of memory space. In this context, we propose a few simplifications in the formal model as constraints:

- We focus on the composition of computation behaviour and the composition of function blocks. Data block composition is not in the scope of the discussion.

- We assume that every data block has only one port. This port can be connected with multiple function blocks. This hypothesis simplifies the connectivity arrays by minimizing the number of ports and port connections.
- Only function block containment is described to reduce the size of the containment array, because scheduler blocks are accompanying with composite function blocks, and data blocks are only involved in the connectivity of composite blocks.
- If a function block contains *one and only one* child function block, the containment can be optionally *simplified* in the formal structural model[1]. Furthermore, as representing composition often requires more memory spaces, it could be helpful to reduce the number of composite function blocks in the structure to minimize the composition efforts and memory usage.

  Nevertheless, the representation of containment is generic and is applicable on any topology that can be expressed by a *rooted tree graph* [49] and a corresponding parent array [53].

Formal Primitives

We define three types of **blocks** to describe an algorithmic computation: **F-blocks** for functions, **D-blocks** for data and **S-blocks** for scheduling, as demonstrated by a *structural model* in Fig. 2.4.

- D-block: contains data that is accessible by F-blocks, denoted as `DB`.
- F-block: performs computation and data processing, denoted as `FB`.

  Each F-block may contain, either a computation function behaving as a **computation function block** (CFB); or a number of child F-blocks, D-blocks and an S-block forming up a **composite computation block** (CPB). The semantics of both CPB and CFB conform to the semantics of F-block.
- S-block: exists only in composite F-blocks, denoted as `S`. It triggers the child F-blocks in a specific order for desired computation behaviour.

External ports are attached at the edges of F-blocks. An F-block may access data stored in a D-block by a **connection** between an external port and the

[1] An example is demonstrated in Section 2.6

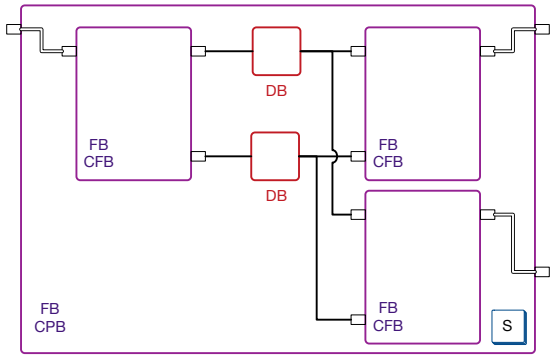

Figure 2.4: Three types of blocks are defined in the structural model: F-blocks for functions, D-blocks for data and S-blocks for scheduling. Ports and D-blocks are connected by connections represented by solid lines; port maps are represented by double solid lines.

D-block. Theoretically, an F-block may be attached with multiple external ports or no external port at all.

Internal ports exist only in composite F-blocks. An internal port is in fact an external port attached on a child F-block. The distinction between internal and external ports is essential to simplify the AOI representation of the connectivity, which will be introduced in later sections. On a composite F-block, an external port is **mapped** with an internal port, so that the internal computations may access external D-blocks. The **port map** is a special type of connection existing only in composite F-blocks.

2.5 Encoding of Computer Readable Formal Model

In this section, we encode the containment, connectivity and scheduling of the formal structural model as computer readable integer arrays using the Computation Behaviour Composition Pattern.

2.5.1 Containment

As depicted in Fig. 2.5, we describe the **containment** relationship of F-blocks using a hierarchical structure. This structure can be abstracted as a *rooted tree graph* shown in Fig. 2.6.

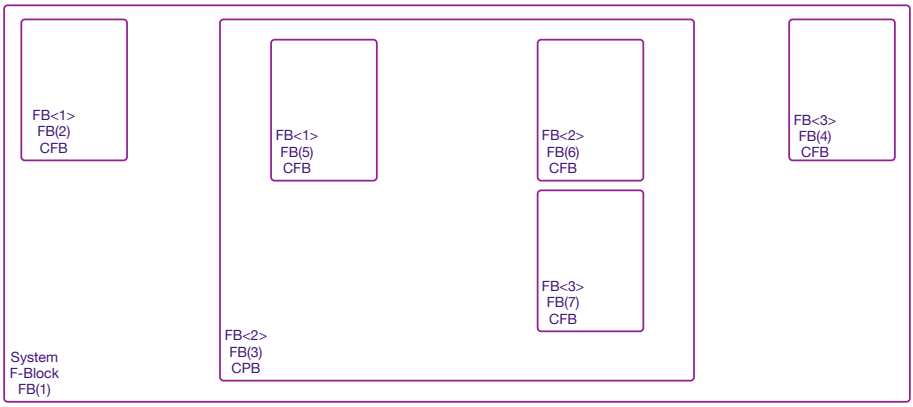

Figure 2.5: A structural model of the containment using CFBs and CPBs.

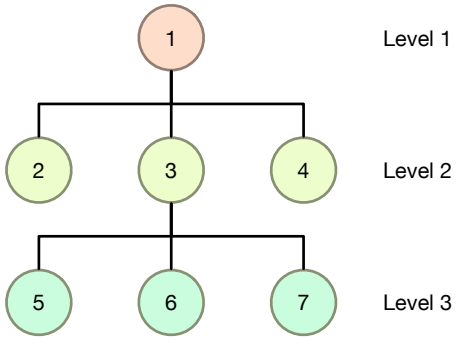

Figure 2.6: The containment of F-blocks can be expressed by a rooted tree graph with levels.

In the tree graph, CFBs are allocated on the leaves, while CPBs sit at branch crotches. The root CPB is a special F-block in the tree, it implies that the composed system can be represented by a composite function block.

We propose the following rule when describing the containment to avoid cross referencing of the F-blocks:

- An F-block can be contained by *one and only one* parent F-block.

This constraint eliminates the risk of creating closed loops in a graph and ensures that the containment can be always represented by a tree.

We define **level** to help describing the containment and to assign F-block IDs. As illustrated in Fig. 2.6, level reflects how far an F-block is away from the root. It is an essential term when analyzing the tree structure to formulate the array-of-integer (AOI) representation.

AOI Representation of Containment

The containment array can be obtained in three major steps.

Step 1: Assigning local IDs for child F-blocks in each CPB

Local IDs are only valid in CPBs. Each child F-block is assigned with a unique local ID in the same CPB.

- Use consecutive natural number starting with 1
- F-block local ID (LID) is denoted as `FB<m>`, $m \geq 1$
- The local ID is unique in a**CPB**
- Local IDs can be assigned arbitrarily in a CPB

Step 2: Assigning global IDs for all F-blocks in the system

Global IDs are valid in the entire system, each F-block has a unique global ID.

- Use consecutive natural number starting with 1
- F-block global ID (GID) is denoted as `FB(n)`, $n \geq 1$
- A global ID is unique in the **system**
- Global IDs should be assigned in the following way: starting from the root CPB in the tree graph, from left to right on the same level, then move on to the next levels

Step 3: Formulating the containment array

We create a **containment array**, denoted as CT, to describe particularly the tree structure using the global IDs assigned in step 2. The values of the CT elements are global IDs.

The containment array is formulated with the following rules:

- The *index* of CT starts from 1, i.e., the first element is $CT(1)$
- If an F-block `FB(n)` is contained by `FB(m)`, $CT(n)$ is set to m
- $CT(1)$ is always 0, indicating that `FB(1)` is the root F-block in the tree and is not contained by any other F-block

Therefore, the containment array of the structure model demonstrated in Fig. 2.5 is formulated as follows:
CT = { 0, 1, 1, 1, 3, 3, 3 }

Interpreting the Containment Array

The containment array formulated above carries all essential information about the hierarchy of the system shown in Fig. 2.5.

Global IDs of CFBs and CPBs
It is straightforward to determine the number of CPBs by observing the non-zero elements in the array. For example, CT indicates that there are two CPBs in the tree, `FB(1)` and `FB(3)`; furthermore, the number of child F-blocks contained in a CPB can be obtained by counting the number of duplicated elements, e.g. CPB `FB(1)` has three child F-blocks since $CT(2)$, $CT(3)$ and $CT(4)$ are all 1; finally, the child F-block IDs can be extracted from the *index* of the array. In the example, `FB(1)` contains `FB(2)`, `FB(3)` and `FB(4)`; `FB(3)` contains `FB(5)`, `FB(6)` and `FB(7)`. The number of CFBs can be calculated by subtracting the number of CPBs from the number of F-blocks. 7 FBs - 2 CPBs = 5 CFBs in this example.

Transit and Level
If two adjacent elements in CT are different, it means a **transit** occurred. A transit implies that the two blocks with adjacent global IDs are contained by different CPBs. For instance, $CT(4)$ and $CT(5)$ are different; `FB(CT(4))`=`FB(1)` and `FB(CT(5))`=`FB(3)`, indicating that `FB(4)` is contained by `FB(1)` and `FB(5)` is contained by `FB(3)`.

An additional check using the global IDs as *element indices* of CT at **transit** is applied to determine whether the transit lead to a **level** change in the tree graph.

As examined above, `FB(CT(4))`=`FB(1)`, and `FB(CT(5))`=`FB(3)`, additional check is then applied on $CT(1)$ and $CT(3)$. In this example, $CT(1)$=0 and

$CT(3)$=1, indicating that there is a level change between `FB(4)` and `FB(5)` since $CT(1)$ and $CT(3)$ are different. It is straightforward to observe the change from the tree graph in Fig. 2.5.

The statistical information that can be extracted from the array are summarized below:

- Number of F-blocks (by the array length)
- Number of CPBs (by counting the non-zero element literal values)
- Number of CFBs (by the difference between the above two)
- Number of child F-blocks in each CPB (by counting the duplicates of the same literal value in the array)
- Number of transits (by transit checking method)
- Number of levels (by level checking method)

In Fig. 2.5:

- There are seven F-blocks in the tree;
- There are two CPBs, `FB(1)` and `FB(3)`, and the rest are CFBs;
  - block `FB(2)`, `FB(3)` and `FB(4)` are contained by block `FB(1)`;
  - block `FB(5)`, `FB(6)` and `FB(7)` are contained by block `FB(3)`;
- There are two transits in the tree, between `FB(1)` and `FB(2)`, and between `FB(4)` and `FB(5)`;
- There are two level changes in the tree, between `FB(1)` and `FB(2)`, and between `FB(4)` and `FB(5)`.

The total number of transits is actually the same as the total number of composite F-blocks, and it is always smaller than or equal to the number of levels.

2.5.2 Connectivity

In this section, we derive the connectivity arrays for the child F-blocks in a composite F-block. Fig. 2.7 illustrates the connectivity of the system in Fig. 2.5.

The following rules are defined for block connectivity:

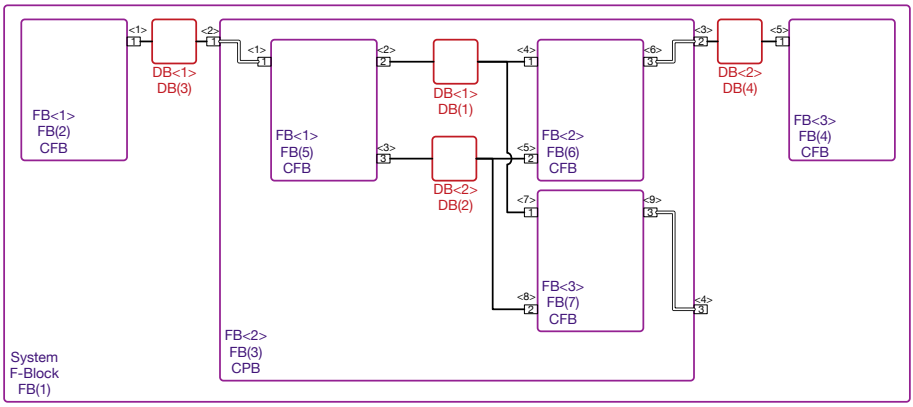

Figure 2.7: D-blocks, ports and connections are added into the system presented in Fig. 2.5. External ports of CFBs and CPBs are represented by their external port IDs on the ports of F-blocks, while internal ports are represented by the local IDs in angle brackets. The double solid lines are port mapping between internal and external ports.

1. An F-block can be connected with n D-block(s) ($n \geq 0$)
2. A D-block can be connected with m F-block(s) ($m \geq 1$)

The first rule regulates the fact that an F-block may run without any data. It is a pragmatic consideration for functions that could run individually without exchanging data with other F-blocks, e.g. a lamp and its on/off switch.

The second rule regulates that the existence of a D-block should be meaningful and it should be involved in the computations at run time. Therefore, a D-block **must** be connected with at least one F-block.

AOI Representation of Connectivity

We advocate four steps to derive the connectivity arrays.

Step 1: Assigning external port IDs on every F-block

- Assign external port IDs on every F-block using consecutive numbers starting with 1
- The external ports are denoted as `EPT(n)`, $n \geq 1$

Step 2: Assigning internal port IDs in every CPB

- In a CPB, assign internal port IDs on the child F-Blocks following the increasing order of the child F-Block IDs
- The internal ports are denoted as `IPT<m>`, $m \geq 1$

Step 3: Assigning local IDs and global IDs for D-blocks in each CPB

Local IDs of D-blocks are only valid in CPBs. Each D-block is assigned with a unique local ID in the same CPB.

- Use consecutive natural number starting with 1
- D-block with local ID (LID) is denoted as `DB<m>`, $m \geq 1$
- The local ID is unique in the **CPB**
- Local IDs can be assigned arbitrarily in a **CPB**

Assigning global IDs (GID) to D-blocks is not always necessary as they are not needed by the connectivity arrays. Nevertheless, we suggest using global D-block IDs in complex systems for convenience. Global IDs of D-blocks are valid in the entire system, we use `DB(n)` to denote a D-block with GID. The global IDs of D-blocks can be assigned arbitrarily in a system.

In Fig. 2.7, there are nine internal ports in `FB(3)`, the internal port IDs are assigned in the following order: The three ports on `FB(5)` are the internal ports of `FB(3)`, denoted as `IPT<1>`, `IPT<2>` and `IPT<3>`. `FB(6)` is the second child F-block of `FB(3)`, the three ports of `FB(6)` are denoted as `IPT<4>`, `IPT<5>` and `IPT<6>`. `FB(7)` is the third child F-block of `FB(3)`, the three ports of`FB(7)` are denoted as `IPT<7>`, `IPT<8>` and `IPT<9>`.

The three external ports of `FB(3)` are `EPT(1)`, `EPT(2)` and `EPT(3)`, corresponding to the one input and two output ports respectively.

Next, we start over the ID assigning procedure for `FB(1)`. `FB(1)` has three child F-blocks, `FB(2)`, `FB(3)` and `FB(4)`. The port on `FB(2)` is `IPT<1>` of `FB(1)`; the three ports on `FB(3)` are `IPT<2>`, `IPT<3>` and `IPT<4>` of `FB(1)`; and the port on `FB(4)` is `IPT<5>` of `FB(1)`.

Step 4: Formulating the connectivity arrays

A connection between a port and a D-block is represented by `[IPT<i>, DB<j>]` with $i \geq 1$, $j \geq 1$.

A connection between an internal port and an external port, known as a port map, is represented by `[IPT<i>, EPT(j)]` with $i \geq 1$, $j \geq 1$.

Every CPB may have a connection array denoted as $CN(n)$, and a port mapping array denoted as $PM(n)$, in which n is the global ID of the CPB. The connection arrays and port mapping arrays are collectively called the **connectivity arrays**.

The connectivity arrays of the system presented in Fig. 2.7 are:

CN(1)={ [1, 1], [2, 1], [3, 2], [5, 2] }

CN(3)={ [2, 1], [4, 1], [7, 1], [3, 2], [5, 2], [8, 2] }

PM(3)={ [1, 1], [6, 2], [9, 3] }

It is worth noting that `IPT<4>` of `FB(1)` is left unconnected and it is not listed in the connectivity array $CN(1)$.

2.5.3 Scheduling

The F-blocks are organized and connected in accordance with the containment and connectivity arrays. Afterwards, computation functions are deployed in the computation function blocks (CFBs), i.e., the *leaf blocks* in the tree graph to compose the computation behaviour.

The D-blocks are assigned with data for the computation, these data are retrieved and updated by F-blocks via connections between F- and D-blocks.

The deployed functions are executed when the containing F-blocks are **triggered**. When triggering a CPB, the embedded S-block will trigger the child F-blocks in the CPB following a **scheduling list** of the S-block, as demonstrated by Fig. 2.8.

A **scheduling list** is a simple array consists of the*local IDs* indicating the triggering order of the child F-blocks in the current CPB. We denote the scheduling list as $SL(n)$, in which n is the global ID of the CPB. For instance, the scheduling list of FB(3) is $SL(3)$. As $SL(3)$={ 2, 3, 1, 2 }, when `FB(3)` is triggered, its child F-blocks will be triggered in the order of `FB<2>`, `FB<3>`,`FB<1>` and `FB<2>` (or `FB(6)`, `FB(7)`, `FB(5)` and `FB(6)`). A child F-block can be triggered multiple times or not be triggered at all. The triggering order is

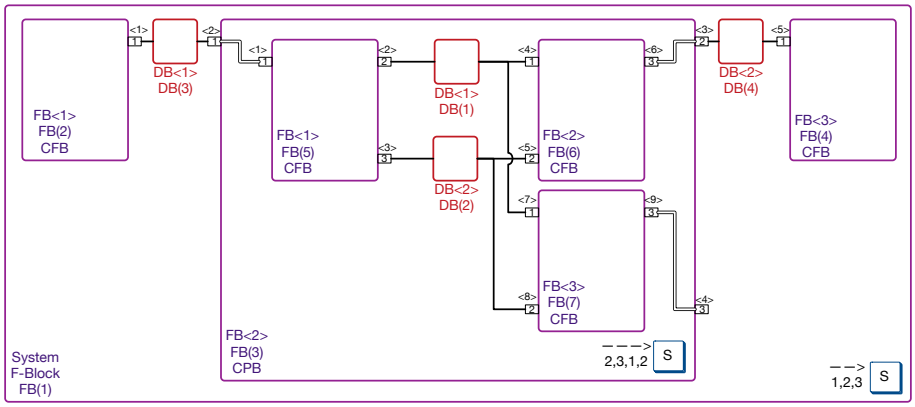

Figure 2.8: Scheduling is introduced in the system to complete the structural composition of behaviour.

configurable to obtain desired behaviours, and it is therefore possible to perform run time rescheduling.

The scheduling list arrays for the structural model in Fig. 2.8 are:

SL(3)={ 2, 3, 1, 2 }

SL(1)={ 1, 2, 3 }

2.5.4 Formalized CPS Meta Model

The CPS meta model is formalized by one containment array, $2m$ connectivity arrays and m scheduling list arrays, where m equals to the number of CPBs. Following the proposed rules, the containment array uniquely flattens the hierarchy of the composition, while the connectivity arrays preciously describe the internal connections inside a composite function block. The scheduling list arrays indicate the triggering orders of child F-blocks in the CPBs, which can be *rescheduled* at run time. As the AOI representations of the CPS involves only computer readable arrays, it is therefore possible to generate the structure of a system using the arrays as “recipes”.

2.6 A Simple Example

For a better understanding of the Cyber-Physical Stack and the formalized structural model, we present a simple example in this section to illustrate how to mode the structure of computation behaviour of a system, and how to create capabilities for a system in the Cyber-Physical Stack context. The desired single task of a setup $\mathbb{G}$ is defined as: detecting the orientation of $\mathbb{G}$ in 3D space.

2.6.1 Conceptual Model

We suppose that $\mathbb{G}$ has a *rigid body*, therefore any point on the setup has the same *angular position* with respect to the *world frame*. We use a three-axis accelerometer to detect linear accelerations in the three perpendicular directions, then calculate *roll* and *pitch* of $\mathbb{G}$.

We consider a microcontroller as a targetable hardware in this example. The peripherals on the microcontroller, for instance general-purpose input/output (GPIO) and inter-integrated circuit (I^2C) are resources on the devices layer (L0) in the Cyber-Physical Stack. In addition to these peripherals, the processing unit and the on-chip memory are also device resources. However, as the CPU and memory are needed by almost every function, they will not be explicitly placed on the devices layer (L0) for simplicity.

The desired task requires a very simple **capability**: `orientation detection`. This capability is composed by three functions in this example: a `sensor driver`, a `matrix multiplication calculator` and a `roll/pitch calculator`, as shown in Fig. 2.9.

The `sensor driver` function handles the data acquisition from the accelerometer. The acceleration is in the form of *rotated gravitational field vector* that can be used to determine the pitch and roll orientation angles [81]. The measured acceleration of $x-$, $y-$ and $z-axis$ from the accelerometer are denoted by X, Y and Z in unit of mg/LSB[2]. The `sensor driver` function must be initialized with the I^2C device address as configuration in data block DB1, and the measured *rotated gravitational field vector* is stored in DB2.

Physically, the accelerometer can be mounted at any place on the rigid body of setup $\mathbb{G}$. When mounting the sensor, it is essential for the setup to be aware of the *sensor's orientation* and link that with the orientation of the setup. In this case, a `matrix multiplication calculator` with a configurable 3-by-3 transformation matrix (stored in DB5) is required. This matrix practically holds

[2] milli-G's per Least Significant Bit, differs on different sensing resolutions.

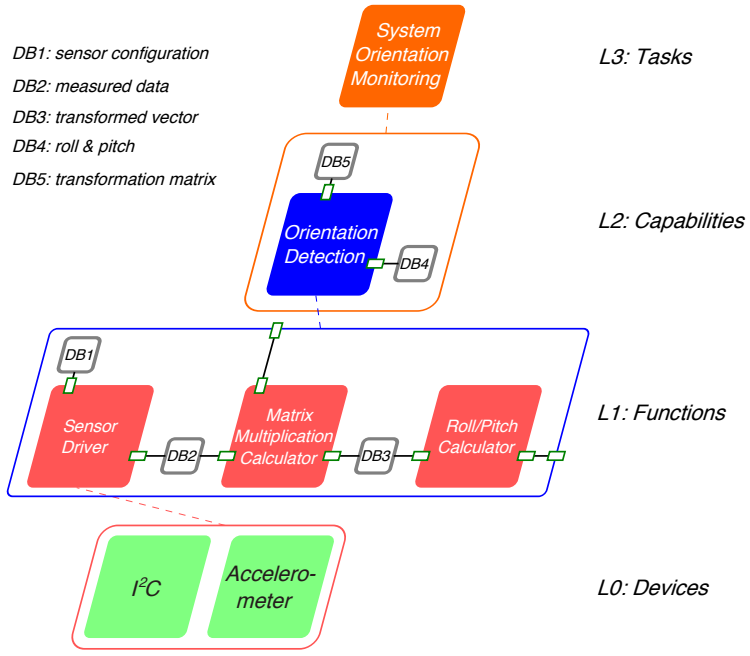

Figure 2.9: A system that has an orientation detection capability modelled with the Cyber-Physical Stack meta model.

the *system-specific* knowledge, which is in this example, the *angle transformation*. The calculator takes the measured data from the accelerometer stored in DB2 and updates the multiplication result in DB3.

The pitch ϕ and the roll θ angles could be determined by Eq. 2.1 and Eq. 2.2 by the `roll/pitch calculator` function. The roll/pitch calculator takes the transformed vector held in DB3 as input and updates the calculated roll and pitch in DB4.

$$\phi = arctan(-\frac{\sqrt{Y^2 + Z^2}}{X})\frac{180}{\pi}, \tag{2.1}$$

$$\theta = arctan(\frac{Z}{Y})\frac{180}{\pi}. \tag{2.2}$$

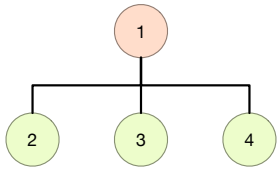

Figure 2.10: Rooted tree graph for the system orientation monitoring task.

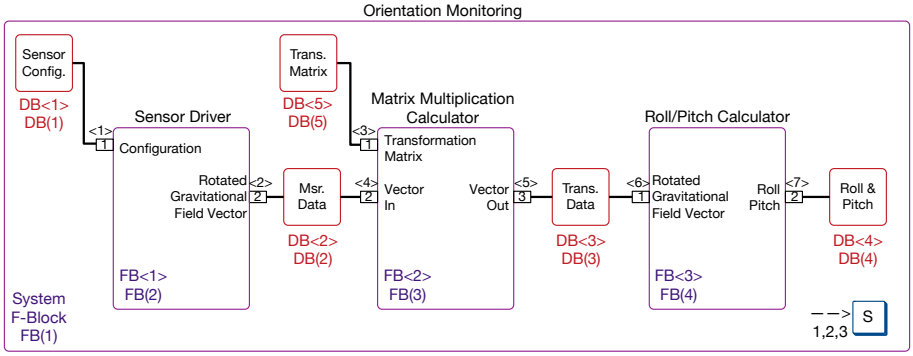

Figure 2.11: The structural model of the orientation monitoring system

2.6.2 Formal Structural Model

Following the considerations of embeddability discussed in Section 2.4, the structure of the conceptual model presented in Fig. 2.9 can be formalized as the one depicted in Fig. 2.11. As the `orientation detection` capability is the only contributor of the task, we neglect the containment on top of this capability for the task, resulting in the rooted tree graph shown in 2.10. The required task is eventually realized by the three functions mentioned in the previous section using three F-blocks.

F-blocks

In Fig. 2.11, `FB(1)` is the system root F-block for task `system orientation monitoring`; `FB(2)` contains the `sensor driver` function; `FB(3)` holds the `matrix multiplication calculator` function; and `FB(4)` is assigned with the `roll/pitch calculator` function. The system root F-block `FB(1)` is a composite function block (CPB) containing three child F-blocks, `FB(2)`, `FB(3)`, and `FB(4)`, which are internally `FB<1>`, `FB<2>`, and `FB<3>`.

D-blocks

The five data blocks in Fig. 2.9 are represented by five D-blocks in Fig. 2.11. According to the consideration of embeddability, the simplification applied to the containment also skips port mapping, therefore `DB(5)` (transformation matrix) in the conceptual model is directly connected with `FB(3)` (matrix multiplication calculator).

S-block
The scheduling of the child F-blocks in `FB(1)` is rather simple, the three F-blocks can be triggered from left to right in Fig. 2.11, i.e. `FB<1>`, `FB<2>`, and `FB<3>` in order to obtain the roll and pitch values. The scheduling order is passed to the S-block as an array of the local IDs.

Connections
All ports on the child F-blocks in `FB(1)` are denoted with an internal local ID. Connections are represented by solid orthogonal lines connecting ports and D-blocks in Fig. 2.11.

2.6.3 AOI Representation

The containment and connectivity arrays as well as the scheduling list are summarized below, following the rules advocated in Section 2.3.

Containment

CT={ 0, 1, 1, 1 }

Connectivity

CN(1)={ [1, 1], [2, 2], [3, 5], [4, 2], [5, 3], [6, 3], [7, 4] }

Scheduling list

SL(1)={ 1, 2, 3 }

2.6.4 Behavioural Concerns

The simple example introduced above demonstrated how to create the structure of computation behaviour of a system using the Cyber-Physical Stack. The

capability is highly **reusable** since it is composed by *system-independent* functions and configured using *system-dependent* knowledge. The simple system in the example is **standalone**, however, it is extensible and may be integrated in a complex system, or system-of-systems. Practically, this standalone system is a **configurable computation** with **configurable scheduling**. In addition to **computation** and **scheduling**, the other concerns introduced in the Composition Pattern, in the form of functions assigned to F-blocks, are also required when turning a standalone system into a reusable sub-system in a complex system, with run time reconfiguration and scheduling capabilities.

Communication
A communication function is required if a sub-system needs to exchange data, events or messages with other sub-systems.

Configuration
A configuration function may access all D-blocks and S-blocks for run time reconfiguration and rescheduling. The configuration function knows *which* D-blocks are being reconfigured, the *knowledge* of the reconfiguration is either pre-defined in the system, or it may come from other systems.

Coordination
A configuration function triggers the configuration function to change the system behaviour, and it is aware of *when* to trigger the reconfiguration.

Monitoring
Monitoring functions are required to observe or raise events with respect to the data exceptions. For instance, a monitoring function may raise a warning event in case the system is upside down according to the detected orientation. The raised event representing an exception and it is captured and processed by the coordinator, and eventually a reconfiguration of the scheduled computations is triggered to resolve the detected exception.

Logging
Logging is an optional concern to record or to show data in D-blocks. A logging function may access the contents of all D-blocks and S-blocks in a system in the Cyber-Physical Stack context. In this example, an LCD can be added into the system as a logger so that the orientation can be displayed at run time. Adding an LCD involves utilizing additional interface such as SPI on the devices layer (L0) and driving functions on the functions layer (L1).

It is worth noting that the above concerns are not necessarily to be implemented in the form of individual functions. For instance, the monitoring function could be shared by different functions to handle events, instead of creating a central monitor; the logging function may be included in several functions to reduce logging delay. Moreover, as `configuration` functions and `logging`

functions have access to **all** D-blocks, we do **not** recommend to connect F-blocks containing these two functions with every D-block in the conceptual model as well as the formal structural model to retain the **readability**.

2.7 Software Architecture

The concern functions mentioned above should be **scheduled** in a reasonable execution order for the desired behaviour of a system, and the scheduling is achieved in a specific **software architecture**. A software architecture is a set of software *structures* plus an *abstraction* of a system [4]. Richards [107] suggested software developers to use formal and well-defined architectures in place toward scalability and reusability, using several common architecture patterns. The development of the Cyber-Physical Stack presented in Section 2.3 is in accordance with the so-called *layered architecture* in [107], but with the important emphasis on the fact that the "layers" are used to bring *structure* in the complexity of the full system and *not* to *hide information* between "layers". Indeed, this section introduces the *event loop meta model*, to execute capabilities in the form of the **scheduled composite computation** for the desired behaviours in the capabilities layer, *while* being reactive to events that can be generated in any of the other layers, e.g., when resource limits are being reached, or task conditions are becoming invalid.

The execution of a modelled *event loop* of a system borrows the idea from *event-driven architectures* as summarized in [107], and this research gives it a *first-class citizen* role as a core building block of composable system-of-systems architectures: it is a conceptual and very simple structural separation between the execution of the behaviour in various concurrently running activities. Various degrees of coupling of such behaviours can also be modelled in somewhat more detail [24], but a more detailed treatment of these aspects are beyond the scope of this research.

2.7.1 Event Loop

The structure of the computation behaviour is represented by containment and connectivity arrays in the formal structural model. The execution of this behaviour is scheduled by *event loop(s)*. An event loop is a time-proven design to couple the functions that compose the capabilities in the CPS meta model into the software architecture in the system implementation. Listing 2.1 gives an example of a typical event loop for single-threaded systems, using the concerns of the Composition Pattern as *primitives*; this meta model is conceptually

Listing 2.1: A typical event loop for single-threaded systems.

```
When the system is triggered
do event_loop{
    communication()
    coordination()
    configuration()
    scheduling()  //of computations
    coordination()
    communication()
    logging()
}
```

extremely simple to understand but yet connects all CPS aspects together in a very structured way, which in itself is fully configurable.

Indeed, every "function" that appears in the event loop is a specific type of computation (possible composite in itself) with all the different purposes modelled in the Composition Pattern. The core of a component's behavioural functionalities are executed in the `scheduling()` function: it triggers all the component's local computations in the composition, including tasks, capabilities and functions. The explicit structure of the event loop meta model allows local computations to take into account what is happening in other concurrently running components, in the basically two only ways in which components can influence each other:

- via *events*, which indicate that "something has happened in the system", to which a local component might have to react to, by means of changing its own behaviour.
- via *data flow*, which provides data from other components that this local component requires for its own functions, or the other way around.

Our meta model let the event loop start with a `communication()` function to receive *data* and *events* from elsewhere (for instance by reading from a communication buffer on hardware), to parse them, and to fill the appropriate local D-blocks, *before* the component starts its own behaviour. This particular choice optimizes the *reactivity* of this component to what is happening elsewhere in the system; a simple *reconfiguration* of this component's event loop model allows other trade-offs between reactivity and local behaviour.

A large number of software frameworks exist, that provide the required inter-process communication [105] with protocols and communication hardware, operating systems and programming languages support, so that the implementation of the `communication()` function poses no pragmatic problems, except maybe for the abundance of choice.

From all the received data, one first inspects the events that were received in the `coordination()` function, because such events might trigger a coordination reaction in the local component. Such coordination action decisions are represented in the form of local *reconfiguration* events, stored in a local D-block, and processed by the next function executed in the event loop, namely the `configuration()` function, which actually carries out the requested reconfiguration.

Since any local computation function in `scheduling()` may generate events and data in itself, it makes sense to execute the local `coordination()` function again (to reconfigure oneself as quickly as possible), before communicating all these events and data to the outside world, in the `communication()` function.

Optionally, the events and data involved in the local computations can be logged by a `logging()` function. In this example, the `logging()` function is at the end of the loop, but in practice, it can be placed at anywhere in the loop if necessary.

The execution condition of the event loop is "when the system is triggered", which is, in accordance with the triggering of the system root block: practically, the event loop is also a `scheduled computation` for desired behaviour, in which multiple capabilities could be scheduled and handled. Multiple event loops may exist in a complex system, in different compositions to realize the desired computation behaviours, and with different trade-offs between reactivity and local behaviour execution.

2.7.2 Scalability in Behaviour Composition

The event loop presented in Section 2.7.1 is a typical example for single-thread systems, especially, it is suitable for embedded devices and has been used on the educational setups introduced in Chapter 6. The event loop formation is in practice **system-specific**. The primitives (or concerns, from the Composition Pattern point of view) are not obliged to be included in a system, e.g. `communication()` is not necessary for standalone systems.

The execution order of the concerns in the event loop as well as the local computations in the scheduled compositions can be significantly different due

Listing 2.2: An event loop example applicable to the fast non-linear moving horizon estimation control algorithm discussed in the work of Vukov [128].

```
When the system is triggered
do event_loop{
  //communication()
  //coordination()
  //configuration()
  scheduling::feedback(){
    computation::dataAcquisition()
    computation::estimatorFeedback()
    computation::controllerFeedback()
  }
  scheduling::preparation(){
    computation::estimatorPreparation()
    computation::controllerpreparation()
  }
  //communication()
}
```

to the desired system functionalities. For instance, a possible event loop for the *fast non-linear moving horizon estimation control algorithm* discussed in the work of Vukov [128] may look like the one demonstrated in Listing 2.2, in the *single-threaded context*.

Fig. 2.12 depicts the division of a control period into preparation and feedback steps, referred from Fig. 2.1 of [128]. The event loop covers the entire control period, which typically starts with a computation gathering data from sensors, i.e. data acquisition, followed by an estimator feedback step and then a controller feedback step. The red arrows imply the *hard constraints* forming up the critical path in the schedule for feedback steps, while the dashed arrows imply the dependency relationship between a feedback step and a preparation step. On a single-threaded system, these steps are **scheduled** as the one shown in Listing 2.2, with possible **extensions** (lines highlighted with "//") such as `coordination()` and `configuration()` for run time self-reconfiguration, or `communication()` for system-of-systems composition.

In practice, the *relationship* between the *primitives* in an event loop, i.e., the scheduling orders of the computations, is determined by the functional requirement and the events involved in the system. Task-, application- and domain-specific knowledge is required by a well-shaped event loop, so that the capabilities and functions may collaborate smoothly to complete the desired

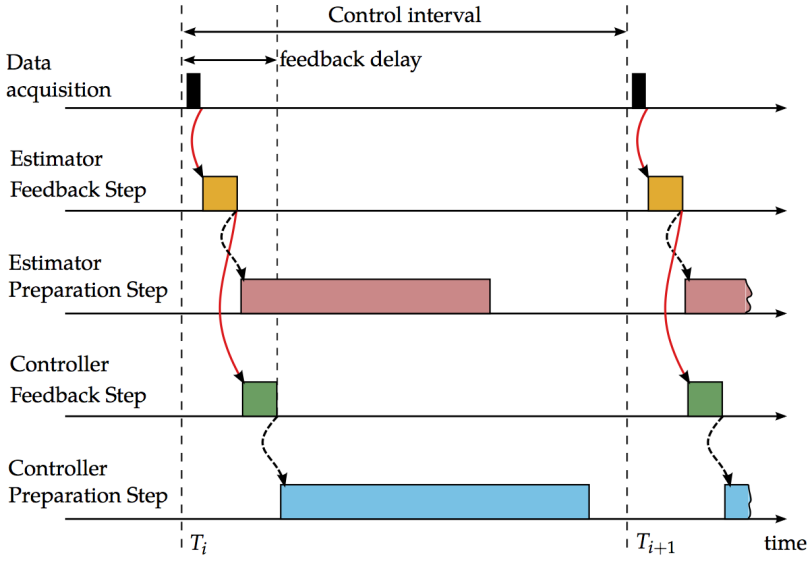

Figure 2.12: This figure is referred from Fig. 2.1 of the thesis by Vukov [128]: division of a control period into preparation and feedback steps.

tasks. For instance, **knowledge of control** is needed in the event loop example demonstrated in Listing 2.2, which determines the *relationship* and brings in dependencies, or *constraints* that influence the scheduling order.

2.7.3 Motion Control Capabilities for System-of-Systems

We borrow the application scenario from Van Parys and Pipeleers, who presented an online motion planning strategy for multiple vehicles in [125], as demonstrated by Fig. 2.13. In this application, four holonomic vehicles are grouped as a formation moving from an initial position to the destination. The vehicles are expected to retain the formation and meanwhile they must be able to avoid static and moving obstacles as a whole. This complex system, which is composed by the four holonomic vehicles, requires a `formation control` capability at run time.

We discuss two typical solutions for the realization of online motion planning using the event loop meta model: distributed formation control, and centralized formation control.

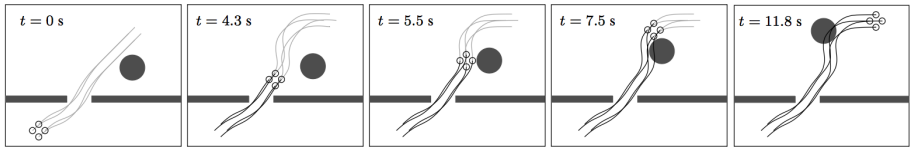

Figure 2.13: This figure is taken from Fig. 3 of [125]. Motion trajectories for a formation of four holonomic vehicles (small circles) in a dynamic environment. The circular obstacle starts moving at t=4s with a velocity of (-0.15, 0.15) m/s. The gray bars represent static obstacles.

Listing 2.3: An event loop that handles an online distributed algorithm for motion planning strategy for cooperating vehicles, adapted from `Algorithm I` in [125].

```
When the system is triggered
do event_loop{
  scheduling::distributedFormationControl(){
    scheduling::localPlanning(){
      computation::followTrajectory()
      computation::estimateFutureStates()
      computation::updateHorizon()
      computation::calculateTrajectory()
    }
    scheduling::interaction(){
      communication::queryStatesOfOtherVehicles()
      computation::planTrajectoryWithStatesConstraints()
      communication::sendLocalStatesToOtherVehicles()
    }
  }
}
```

Distributed Formation Control

In the distributed solution as discussed in [125], the vehicles plan their trajectories individually to retain the formation while avoiding static and moving obstacles. The presented online distributed algorithm running on each vehicle fits well in the event loop meta model. We adapt the steps in the algorithm as scheduled computations as shown in Listing 2.3.

At run time, each vehicle plans its own trajectory in `scheduling::localPlanning()`

Listing 2.4: An event loop that handles the centralized motion planning algorithm for cooperating vehicles.

```
When the system is triggered
do event_loop{
  communication::switchRoleIfRequired()
  coordination::handleEvents()
  configuration::configureCapabilities()
  scheduling::centralizedFormationControl(){
    communication::queryStatesFromAll()
    scheduling::planning(){
      computation::estimateFutureStatesForAll()
      computation::updateHorizon()
      computation::calculateTrajectoriesForAll()
    }
    communication::sendTrajectoriesToAll()
  }
  computation::followTrajectory()
}
```

first, and then use the states of the other vehicles as additional constraints to optimize this trajectory. Communication functions are needed in `scheduling::interaction()` in order to query/send states from/to other vehicles.

Centralized Formation Control

The same application can be realized by centralized motion planning as well, as presented in Listing 2.4. We suppose one of these vehicles takes the role of *central planning vehicle* that calculates and sends the desired set points to all the other *members*. At any moment, the vehicles may negotiate and switch roles, if the current central planning vehicle is not available any more.

The event loop in Listing 2.4 applies to all the vehicles in the centralized solution. At run time, the first `communication()` function monitors the received events and data in the message buffer for desired set points and operation mode. The `coordination()` function gets the event raised by `communication()` and triggers the `configuration()` function to *reconfigure* the computations involved in the loop (e.g. maximum speed), and to *reschedule* the loop for different behaviour: either activating the centralized motion planning function

`scheduling::centralizedFormationControl()` or not, with respect to the role of a vehicle in the formation.

The `scheduling::centralizedFormationControl()` function holds the essential steps for the motion planning of the formation, including additional `communication()` functions to query states and to send new set points to other vehicles in the system.

2.8 Conclusion

This chapter introduced the conceptual and formal meta models of the Cyber-Physical Stack, aiming at providing a systematic methodology of composing computation behaviour for cyber-physical systems and system-of-systems, on which self-reconfiguration and run time recomposition are supported.

The formalization focused on the *structure* of algorithmic computation of the CPS meta model, using the NPC4 primitives for structural composition. A Computation Behaviour Composition Pattern, as the **first major contribution** of this chapter, is developed and used in guiding the encoding of the computer readable formal model.

The computer readable formal structural model together with the encoding approach (in the form of AOI representation) for containment, connectivity and scheduling of computation behaviour is the **second major contribution** of this chapter. First of all, the modelling of containment is applicable on any rooted tree graph representable containment, as long as the rules are strictly complied with. The same approach is used in Chapter 3 to formulate the containment array of composite states and state machines. Secondly, as the containment, connectivity and scheduling are expressed by compact computer readable, non-negative integer arrays, it is thus feasible to *automate* the structural composition of computation behaviours using these arrays as scripts with significant composability and reusability. Next, the *embeddability* of the encoded formal model is obvious: the non-negative integers are practically of *unsigned integer type* in many programming languages, which is therefore optimal for embedded computing and storage. Lastly, as the F-, D- and S-blocks are *containers*, the *cargoes* carried by these containers, i.e., functions, data and schedules, can be replaced at run time. This feature consequently improved the system *flexibility* and *reusability*, and eventually guided the development of the composable and embeddable software framework discussed in Chapter 4.

In this chapter, we proposed a sharper model of *system capability* to overcome the weakness mentioned in the state of the art as the **third major contribution**.

We advocate that a capability is composed by *system-independent* functions using *knowledge* as configuration parameters. These parameters could be system-, environment- or task-specific, extra human efforts and computation resources are therefore required by a *system-dependent* capability, such as the transformation matrix for `orientation detection` capability discussed in Section 2.6. The *separation* of system-dependent and system-independent aspects significantly improved the reusability of functions and capabilities, the composability of system behaviour, and the flexibility of reconfiguration as well as rescheduling, with an acceptable and reasonable "cost" on computation resources paid for the additional formalized structure of computation behaviour. The above-mentioned extra cost can be minimized[3] with the consideration of embeddability discussed in Section 2.4.

In addition to "how-to create" capabilities, we also propose a concrete solution for "how-to use" them, by means of the **event loop** meta model as a particular type of software architecture, which reveals and guides the scheduling of functions involved in capabilities. The event loop provides a complementary view of computation behaviour in the CPS meta model, it helps system designers to determine how to merge capabilities in a system, and it also facilitates the behavioural integration in system-of-systems using additional Composition Pattern concerns as **extensions** in the event loop, as demonstrated in Section 2.7.

There are two major limitations to overcome in the developed methodology. First, the model of scheduling list clearly presents the execution order of child F-blocks in a composite F-block (CPB), however, the proposed scheduling function is rather simple. In a large system, the triggering orders could be complex and the scheduling lists might by frequently modified for desired functionalities and system performance. Secondly, the behaviour of communication between sub-systems in complex system is not modelled. A well-structured behaviour model of communication could improve the efficiency when integrating standalone systems into system-of-systems, and further reduce the efforts needed by implementing communication protocols. In the future work, we would like to invest sufficient efforts in the above-mentioned scheduling and communication behaviour models, to enhance the behavioural modelling aspect of the Cyber-Physical Stack.

For the implementation and demonstration of the orientation detection setup discussed in Section 2.6, please visit: `https://gitlab.mech.kuleuven.be/u0066910/ces`

[3]The same cost can also be quantified in unit of bytes. An example of the resource consumption in C implementation on microcontrollers is discussed in Section 4.8

Chapter 3

Composition of Coordination and Life Cycle State Machine

This chapter introduces a *composable Finite State Machine* (cFSM) meta model to facilitate the coordination in an autonomous self-reconfigurable cyber-physical system. This meta model guided the development of a system-level coordinator model, the *Life-Cycle State Machine* (LCSM). The major property of the LCSM is a clean separation of "what can be used", i.e. the *resources*, and "what can be achieved" i.e. the *capabilities*.

Coordination defines the execution and interaction semantics of functional computations [65], and it is essential for complex systems to regulate the computation behaviours by determining how the sub-systems effectively collaborate with each other. Computation behaviours are influenced by their configurations. The Coordination-Configuration Pattern advocates a best practice of *pure coordination* [66] using state machines as event processors to determine states and raise events. Pure coordination helps to avoid platform specific dependencies and therefore prevents blocking invocations of operations, as it explicitly decouples computation and coordination.

Vanthienen et al. [127] introduced a Life-Cycle Finite State Machine in constraint-based task specific robot applications [26] to coordinate robot tasks. The Life-Cycle Finite State Machine model is based on the *task level lifetime semantics* proposed by Soetens [119], the Coordination-Configuration Pattern and the reduced Finite State Machine (rFSM) [68]. As one of the major contributions of this chapter, the Life-Cycle Finite State Machine is extended from *task level* to *system level*, and is named as Life-Cycle State Machine

(LCSM) which reveals the **resource-capability relationship** in behaviour coordination. The meaning of "capability" in this chapter is different from the one in the CPS meta model: resource-capability reflects the relationships between *causes* and *consequences*, and between *purposes* and *constraints*.

In the next sections, we will discuss the conceptual and formal meta model of the composable Finite State Machine (cFSM), and use the formalized cFSM meta model to create concrete model of the Life-Cycle State Machine.

3.1 Composable Finite State Machine: Conceptual and Formal Models

The concept of the cFSM is consistent with the*reduced Finite State Machine* (rFSM) presented by Klotzbücher et al. in [68]. The rFSM introduced a minimal variant of Harel statecharts [50] to model the coordination of robot tasks and systems using the following primitives: **states, transitions** and **connectors**. These primitives are inherited by the cFSM, and we use **containment** and **connectivity** to describe relationships. In the cFSM, we introduce additional focus on the *composability* and apply necessary simplifications and constraints for the *embeddability*.

A **state** can be, either a *composite state* or a*leaf state* according to the definition of UML2 and Harel statecharts. As introduced in [50], states can be hierarchical, implying that the type of a state is dependent on whether it contains child states or not. In this context, a composite state can be considered as a *state machine* in which multiple child states are embedded. The composite state on the top-level of the hierarchy is denoted as the **root** state.

Fig. 3.1 demonstrates a simple state machine with all the above primitives. The two ellipses are **leaf states** (LST); the round corner rectangle represents a **composite state** (CST) containing the two leaf states, it is also the **root state** in the example.

We propose the following constraints in the cFSM meta model for **states**:

1. A state can be contained by *one and only one* parent state
2. *One and only one* **leaf state** is **active** at any moment, known as the **current state**
3. The **composite state** who contains the active **leaf state** is **active**

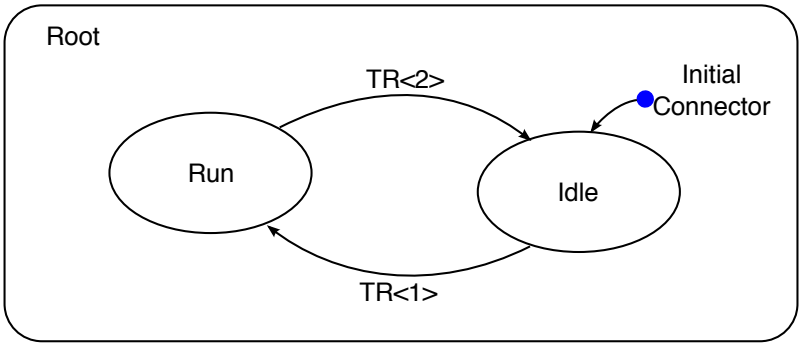

Figure 3.1: A simple composable finite state machine. The two ellipses are leaf states; the round corner rectangle represents a composite state, and it is also the root state in this state machine. The curved arrows are transitions, and The filled circle is an initial connector indicating the entry state of the composite state.

4. The composite state who contains the **active composite** state is also **active**

The constraints above permits representing the entire **active state** of a state machine by means of a single **leaf state** [68], and they are consistent with the W3C State Machine Notation [129] which requires that*for any active state, all parent states are active too.*

A **transition** is represented by a single curved arrow, leaving from a *source* state and pointing to a *destination* state. If the source state of a transition is active, **triggering** this transition will *deactivate* the *source state* and *activate* the *destination state*, otherwise the triggering action is ignored. For instance in Fig. 3.1, if the `Idle` leaf state is active, triggering transition `TR<1>` will deactivate `Idle` and activate `Run`. The current state is therefore changed from `Idle` to `Run`.

We define two types of transitions in the cFSM. **External transitions** are the *outgoing* transitions of a state, i.e., a state is the source state of all its external transitions; and **internal transitions** are the transitions that are embedded in a composite state, which exist only in composite states.

A transition could be triggered by one or multiple **events**. Whether a transition is triggered or not depends on the current state and the **events**. The event-transition function is defined by Eq. 3.1, in which TR is the enabled transition,

EV_i and EV_o are input and output events, and ST_c is the current state.

$$[TR, EV_o] = f(ST_c, EV_i) \tag{3.1}$$

The filled circle is an **initial connector**, it is unique in a composite state. The initial connector is the entrance of a composite state, indicating the first child state to activate. The transition connected on an initial connector is automatically triggered once when the composite state is activated.

3.2 Encoding of Computer Readable Formal Model

Similar to the formal modelling of the computation structure in the CPS meta model, we use the array-of-integer (AOI) representation to handle the structural composition of the cFSM meta model, using a set of containment and connectivity arrays. The operation of a state machine is determined by triggering the transitions using events, the relationship between events and transitions could be described by an application specific *event-transition table*.

3.2.1 Containment

Fig. 3.2 is a structural model of a state machine. The **containment** of states can be expressed in a hierarchical structure as a **rooted tree graph** [49], in which the leaf states are located on the leaves of the tree while the composite states take positions at the branch crotches, as shown in Fig. 3.3. The root state is the root of the tree.

We propose the following rule when describing hierarchical state machines as a constraint, to prevent creating close loop in the tree graph:

- A state, either leaf or composite, can be contained by *one and only one* parent composite state.

We define **level** to help describing the containment and numbering the states. The level information of the states can be obtained from the tree graph, as indicated by Fig. 3.3.

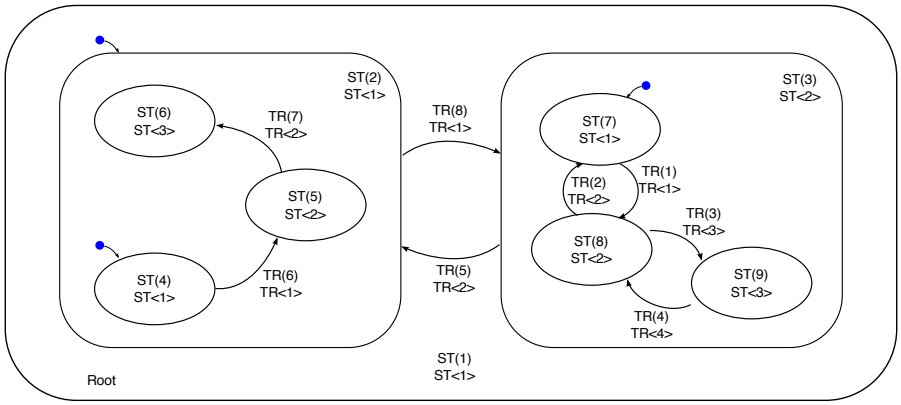

Figure 3.2: A state machine example.

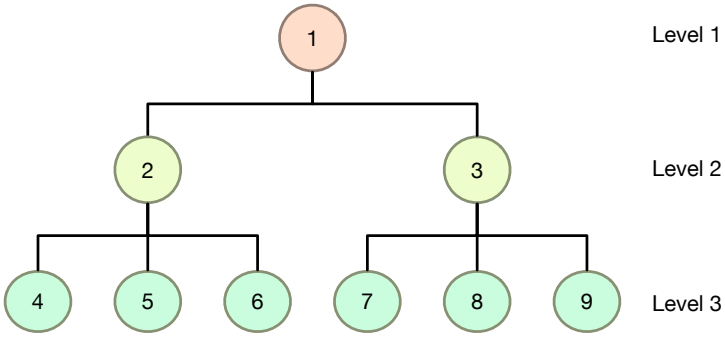

Figure 3.3: The containment tree graph of the state machine in Fig. 3.2

AOI Representation of Containment

To obtain the AOI representation of the containment, we must assign global IDs and local IDs to the states as we did in Section 2.5.1 for the F-blocks. We advocate three steps to derive the containment array.

Step 1: Assigning local IDs to states for child states in each composite state

Local IDs are only valid in CSTs. Each child state is assigned with a unique local ID in the same CST.

- Use consecutive natural number starting with 1
- State with local ID (LID) is denoted as `ST<m>`, $m \geq 1$
- The local ID is unique in a composite state **CST**
- Local IDs can be assigned arbitrarily in a CST

Step 2: Assigning global IDs for all states in the system

Global IDs are valid in the entire state machine, each state has a unique global ID.

- Use consecutive natural number starting with 1
- State with global ID (GID) is denoted as `ST(n)`, $n \geq 1$
- A global ID is unique in the **state machine** or **root state**.
- Global IDs should be assigned in the following way: starting from the root state, from left to right on the same level, then move on to the next levels

Step 3: Formulating the containment array

The **containment array** is denoted as CT, it describes the tree structure using the global IDs assigned in Step 2. Elements in CT are global IDs of the states, and the array length is the total number of states.

The containment array is formulated with the following rules:

- The index of CT starts from 1, i.e., the first element is $CT(1)$;
- If a state `ST(n)` is contained by `ST(m)`, $CT(n)$ is set to m;
- $CT(1)$ is always 0, indicating that `ST(1)` is the root state in the tree and is not contained by any other state

Therefore, the containment array of the state machine shown in Fig. 3.2 is formulated as follows:
$CT = \{\ 0, 1, 1, 2, 2, 2, 3, 3, 3\ \}$

The containment array could be interpreted in the same way by defining **transit and level**, as those introduced in Section 2.5.1. As a summary, the following information can be extracted from the containment array of the state machine in Fig. 3.2:

- There are nine states in the tree;
- There are three CSTs, `ST(1)`, `ST(2)` and `ST(3)`, and six LSTs;
  - State `ST(4)`, `ST(5)` and `ST(6)` are contained by state `ST(2)`;
  - State `ST(7)`, `ST(8)` and `ST(9)` are contained by state `ST(3)`;
  - State `ST(2)` and `ST(3)` are contained by state `ST(1)`;
- There are three transits in the tree, between `ST(1)` and `ST(2)`, `ST(3)` and `ST(4)` and between `ST(6)` and `ST(7)`
- There are two level changes in the tree, between `ST(1)` and `ST(2)`, and between `ST(3)` and `ST(4)`

The total number of transits is the same as the total number of composite states, and it is always smaller than or equal to the number of levels.

3.2.2 Connectivity

The state connectivity in the cFSM meta model is similar to the block connectivity explained in Section 2.5.2. The only difference is that a connection in the cFSM involves one transition and two states.

The following rules are applied in defining the state connectivity with the *consideration of embedded-friendliness* in order to reduce memory usage:

1. A state should be connected with n transition(s) ($n \geq 1$)
2. A transition must have exactly **one** *source* state and **one** *destination* state
3. Every state should be **reachable**

The first rule implies that an isolated state without any transition is not allowed in a state machine, as this state will never be activated. The second rule ensures that every state is connected by a transition, and every transition has a source state and a destination state. The last rule prevents isolated subgroup of connected states, as shown in Fig. 3.4. State machines that violating the above rules are considered as *invalid*.

We advocate that either isolated state or isolated state subgroup should be avoided to keep the model compact, as the isolated states are not reachable and will introduce redundancy and extra effort on modelling and implementation. The requirement of embeddability also triggers us to simplify the model and the implementation to efficiently and effectively utilize the available hardware devices and resources.

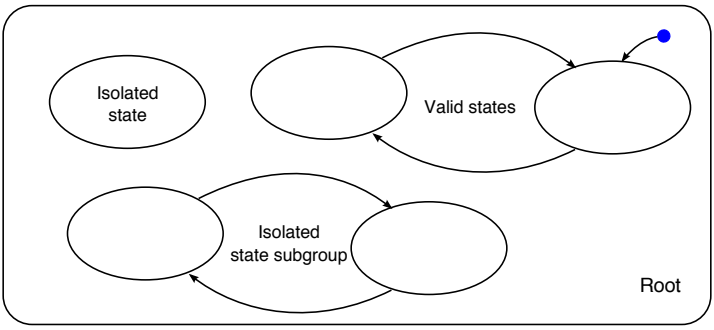

Figure 3.4: An invalid state machine. Either isolated state or state subgroup should be avoided to (1) keep the model compact, (2) to minimize the modelling and implementation effort, and (3) to maximize resource utilization efficiency when deploying the implemented model on hardware.

AOI Representation of Connectivity

The connectivity arrays can be obtained in two major steps.

Step 1: Assigning transition local IDs and global IDs in every composite state

The transition local IDs are only valid in the same composite state.

- Use consecutive natural numbers starting with 1
- Follow the order of child state local IDs
- Assign transition local IDs only for the *outgoing* transitions of each child state
- Transition local IDs are denoted as `TR<m>`, $m \geq 1$

The method in this step is inspired from the following facts:

1. The child states in a composite state and the transitions can be modelled by a directed graph, as every transition has a direction
2. A directed graph can be expressed by an adjacency list by describing vertices and their corresponding outgoing edges [38]

Assigning global IDs (GID) to transitions is not a must as they are not needed when constructing the state machine structure from the connectivity arrays. Nevertheless, we suggest using transition global IDs in complex state machines to help specifying transitions on different containment levels. The global IDs of the transitions can be arbitrarily assigned, denoted as `TR(n)` in Fig. 3.2.

Step 2: Formulating the connectivity arrays

In the cFSM, connections are represented in the form of `[TR<i>, ST<j>]`, $i \geq 0$, $j \geq 1$, in which `TR<0>` represents the initial connector of each composite state, and the entry state might be any child state contained in the composite state.

The connections in the state machine is therefore listed as:
CN(1)={[0, 1], [1, 2], [2, 1]}

CN(2)={[0, 1], [1, 2], [2 ,3]}

CN(3)={[0, 1], [1, 2], [2 ,1], [3, 3], [4, 2]}

3.2.3 State Machine Operation

In the cFSM, a state machine is operated by triggering the transitions using events. The event-transition function in Eq. 3.1 introduced in Section 3.1 is actually a model of an **event handler**.

Taking the current state ST_c and the input event EV_i as independent variable, the function triggers a transition TR and may optionally raise an output event EV_o if required.

Boundary Crossing Transitions

A boundary crossing transition is defined as a transition connecting two states by crossing the boundaries of composite states as shown in Fig. 3.5.

The state machine in Fig. 3.5 is extended from the one in Fig. 3.2, in which transitions `TR(5)` and `TR(8)` are both boundary crossing transitions.

In the current cFSM meta model, boundary crossing transitions are not explicitly modelled as a major limitation. Nevertheless, this type of transitioning in a state machine can be realized by adding an entry in the event-transition table, to virtually connect the two states involved in a boundary crossing. For instance, in Fig. 3.5, the triggering condition of transition `TR(8)` could be: `TR(8)` is

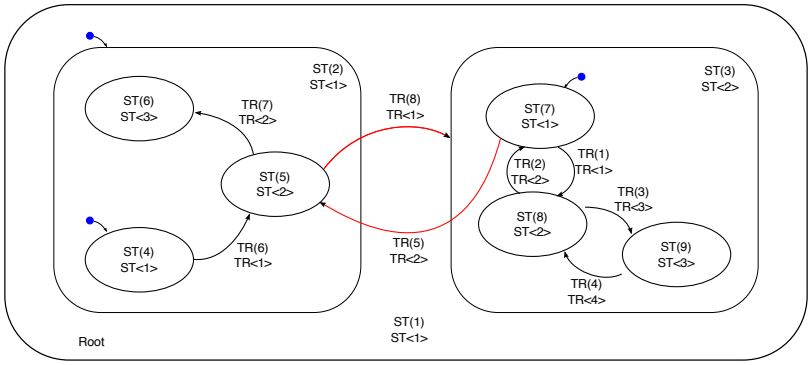

Figure 3.5: Boundary crossing transitions connect two states by crossing the boundaries of composite states.

enabled when the leaf state `ST(5)` is the current active state. Similarly, the triggering condition of transition `TR(5)` could be: `TR(5)` is enabled when the leaf state `ST(7)` is the current active state, and `TR(6)` is enabled immediately afterwards. In this case, the structural constraint is partially handled by behavioural techniques for the desired result.

In practice, the event-transition function can be complex due to sophisticated transitioning in state machine operation. We leave the triggering mechanism for future work as it is out of the scope of this thesis.

3.3 Life Cycle State Machine

This section discusses a concrete coordination model, *the Life-Cycle State Machine* (LCSM). The LCSM is consistent with the Coordination-Configuration Pattern aiming at improving the coordination reusability [66]. In addition, the LCSM introduces **resource** and **capability** to differentiate *what can be used* and *what can be achieved* into the Coordination-Configuration Pattern: the behaviour of a capability is determined by the configuration of the corresponding resources, and the implementation of a behaviour is deployed on the corresponding resources.

The resource-capability relationship covers a broader scope beyond the literal meaning of the two terms. For instance, in the CPS meta model, software functions are resources that can be used to realize certain capabilities, and meanwhile, capabilities are also resources for tasks. This relationship extends the

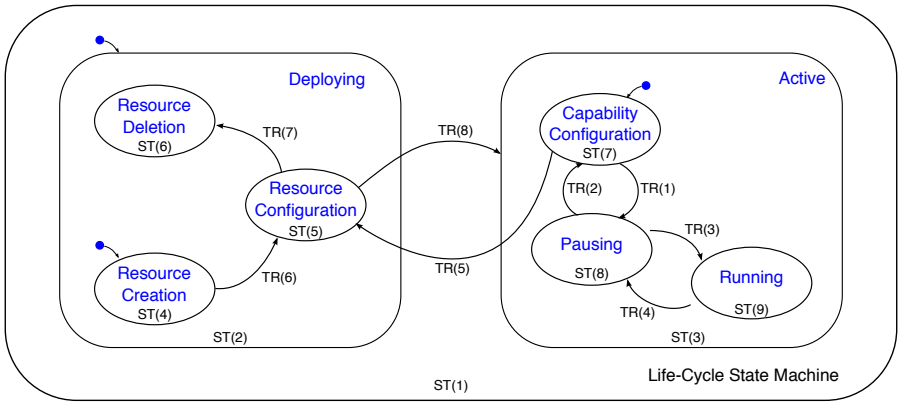

Figure 3.6: The Life-Cycle State Machine model.

scope of the Coordination-Configuration Pattern from task level to system level, and it also implies that the Coordination-Configuration Pattern is applicable between the software layers of the Cyber-Physical Stack meta model.

3.3.1 Conceptual Model of Life-Cycle State Machine

The conceptual model of Life-Cycle State Machine (LCSM) is demonstrated in Fig. 3.6. The primitive states are: `Deploying`, `Active`, `Resource Creation`, `Resource Configuration`, `Resource Deletion`, `Capability Configuration`, `Pausing` and `Running`. The relationships and constraints are reflected by the transitions between the states and the triggering condition of these transitions.

We define two composite states in the LCSM: CST `Deploying` for resources and CST `Active` for capabilities. The idea is straightforward and is valid in most of the robotic and cyber-physical systems: capabilities become available only when they are completely deployed on the available resources. The `Deploying` state covers the stages of resource preparation including creation, configuration and deletion, while the `Active` state covers the configuration, pausing and running of these desired capabilities.

At start up, the LCSM enters the `Deploying` state, and then the `Resource Creation` leaf state, as indicated by the initial connectors. Resource creation usually refers to claiming certain resources instead of physically creating them. Once the resources are created, they should be configured in the `Resource Configuration` state to ensure that the behaviours of these resources meet the

requirement of the desired capabilities. At this moment, the resources are ready for the capabilities. Nevertheless, we introduce the `Resource Deletion` state to keep the life-cycle complete: The claimed and configured resources could be released in this state, and the LCSM will stay in this state and wait for a restart.

When the resources are successfully configured in `Resource Configuration` state, the LCSM is ready to enter the `Active` state. The entry state of `Active` is the `Capability Configuration` state indicated by the initial connector. In this state, the capabilities are configured for the overall desired behaviour. We advocate a `Pausing` state between `Capability Configuration` and `Running` because `Pausing` explicitly indicates that capabilities are **actively paused** and are not used by any other system or client; while `Running` implies that the capabilities are used by other systems and is in service. For instance, in the multi-robot system example, a robot could be in the `Pausing` state to wait for other robots motion. While being actively paused, it will not move but still hold its own position and keep the joint torques activated.

Sometimes it is necessary to switch from the `Active` state back to the `Deploying` state to reconfigure the resources, or to terminate the life-cycle by deleting the resources. We use a boundary crossing transition leaving from `Capability Configuration` and entering the `Resource Configuration` to realize the switching, so that the resources can be reconfigured or released by transitioning to the `Resource Deletion` state.

3.3.2 Formalization of the LCSM Model

The LCSM model is formalized by one containment array, three connectivity arrays and an event-transition table.

To formalize the LCSM model, we need to determine the containment and connectivity arrays first. The structural composition of the LCSM model is exactly the same as the one presented in Fig. 3.5. The containment and connectivity arrays are already obtained in the formalization example in the previous section:

CT = { 0, 1, 1, 2, 2, 2, 3, 3, 3 }

$CN(1)$={[0, 1], [1, 2], [2, 1]}

$CN(2)$={[0, 1], [1, 2], [2 ,3]}

$CN(3)$={[0, 1], [1, 2], [2 ,1], [3, 3], [4, 2]}

Next, we need to specify the event-transition table and create an event handling function to operate the LCSM. The entries in the event-transition table determine the behaviour of the LCSM. In the current LCSM model, every transition is triggered by a single event.

The entries are listed in Table 3.1.

Table 3.1: The event-transition table of the LCSM model

Trans. ID	Leaf State	Transition	Event
TR(1)	Capability Configuration	TR_capconf_pau	EV_capconf_pau
TR(2)	Pausing	TR_pau_capconf	EV_pau_capconf
TR(3)	Pausing	TR_pau_run	EV_pau_run
TR(4)	Running	TR_run_pau	EV_run_pau
TR(5)	Capability Configuration	TR_act_dep	EV_act_dep
TR(6)	Resource Creation	TR_cre_resconf	EV_cre_resconf
TR(7)	Resource Configuration	TR_resconf_del	EV_resconf_del
TR(8)	Resource Configuration	TR_dep_act	EV_dep_act

Boundary crossing transition `TR(8)` could be triggered by event `EV_dep_act` only when the current leaf state is `Resource Configuration`. As a consequence, `Resource Configuration` and `Deploying` states will be deactivated in succession before the activation of `Active` and `Capability Configuration`.

The other boundary crossing transition `TR(5)` is more complicated as its destination `Resource Configuration` is not the entry state of the composite state `Deploying`. Additional operations are therefore required to realize boundary crossing. The state machine handler first triggers `TR(5)` to enter the `Deploying` state, and immediately trigger `TR(6)` to switch the current leaf state to `Resource Configuration`; or we may alternatively change the entry state of `Deploying` to `Resource Configuration` after the resources are claimed, as `Resource Creation` is needed only at the beginning of the life-cycle.

3.4 Conclusion

This Chapter introduced the composable Finite State Machine meta model to systematically model and create state machines for computation coordination in cyber-physical systems. The structural composition of the cFSM meta model is formalized using the rFSM concepts and primitives, and is encoded in a computer readable form by containment and connectivity arrays that can be reused in state machine construction.

The cFSM meta model is composable and embeddable. It permits recomposition of existing models with each other as "states containing other states" is allowed; the compactness of the arrays is an embedded-friendly and necessary feature for microcontroller based applications.

On the second half of this chapter, a concrete coordinator model, the Life-Cycle State Machine (LCSM) is proposed and created using the cFSM meta model. The LCSM explicitly separates "what can be used" (resources) and "what can be achieved" (capabilities), which extended the previously developed Life-Cycle Finite State Machine [127] from task level to system-wise. It is a suitable coordinator for the coordination between *functions layer* (L1) and *capabilities layer* (L2), and between *capabilities layer* (L2) and *tasks layer* (L3) in the CPS meta model.

The cFSM meta model conforms to the NPC4 meta model. However, to pursue the simplicity for the embedded-friendly feature, the cFSM model did not use **port** as one of the primitives. As a trade-off, it is difficult to explicitly model boundary crossing transitions, they have to be alternatively realized by interfering the behavioural inputs, i.e. the event-transition table and the event handler, as the one in the Life-Cycle State Machine, discussed in Section 3.3. The modelling of boundary crossing transition in the cFSM will be covered in the future work.

Chapter 4

Cyber-Physical Stack Software Framework

This chapter discusses the implementation of the Cyber-Physical Stack (CPS) software framework. This work was initiated by research objective 2 and 3: *Developing a composable and embeddable software framework for the CPS approach.*

The CPS software framework is implemented in C language and is independent from platform specific libraries. With the cross-platform feature, it can be compiled for general purpose processors like Intel, ARM and PowerPC, as well as embedded microcontrollers such as Atmel and Microchip devices. The implementation is guided by the formalized and encoded structure of CPS meta model developed in Chapter 2.

In the next sections, we will focus on the programming ideas and thinking, by briefly introducing some *C structs* and *function definitions* in the software framework. The goal of this chapter is to provide an overview of how the composition of computation behaviour could be realized, instead of getting down to the coding details.

4.1 Model of Implementation

We use the encoded formal structure of the CPS meta model introduced in Section 2.5 to guide the implementation by separating structural composition

from behaviours. The containment and connectivity arrays are the keys to create the system structure using F-, D- and S-blocks, and the system behaviour is determined by the functional *computation* assigned in F-blocks and the *scheduling* of these F-blocks.

To compose a *system* we need to:

- Compose the structure using the array-of-integer (AOI) representation of a structural model, i.e. the containment and connectivity arrays as recipes
- Assign functions to the computation function blocks (CFBs)
- Assign data to the data blocks
- Assign scheduling list to the scheduling blocks in composite computation blocks (CPBs)

Run time reconfiguration is realized by modifying the data stored in the corresponding D-blocks or by replacing the functions assigned in F-blocks, and run time rescheduling is achieved by changing the scheduling list of the S-blocks.

4.2 Composition Descriptor

Fig. 2.8 intuitively illustrates a system structure using F-, D- and S-blocks. The relationships between the blocks are encoded in the containment and connectivity arrays, and therefore are interpretable by software programs. However, these arrays do not contain the structural description of F-blocks, for instance the number of attached ports. In addition, it is crucial to be aware of the numbers of child F-blocks, D-blocks and internal connections contained in a composite F-block when creating the system structure to avoid unexpected memory leaks in practice. Moreover, it would be handy if additional information about an F-block, either textual or literal, could be retrieved at run time to facilitate block identification.

With the above consideration, we propose a **composition descriptor** to keep the structural information for F-blocks:

- Metadata

  Metadata is structured information that describes, explains, locates, or otherwise makes it easier to retrieve, use, or manage an information resource [91].

- Block status

  The block status indicates whether a block is fully composed or not. A computation function block (CFB) is fully composed when:

  1. All ports are attached to the CFB
  2. A computation function is assigned to the CFB

  A composite computation block (CPB) is fully composed when:

  1. All ports are attached to the CPB
  2. All embedded D-blocks are assigned to the CPB
  3. All connections are connected
  4. All child F-blocks are fully composed
  5. The scheduling list is assigned

- Number of ports

  Total number of external ports of an F-block.

- Numbers of connections, D-blocks and child F-blocks

  Numbers of connections, D-blocks and child F-blocks in a composite computation blocks (CPBs), only valid for CPBs.

- Numbers of **registered** ports, connections, D-blocks and child F-blocks

  These are counting variables used during the structural composition to check F-block completeness.

- F-block name

  An optional entry to store a textual name for F-block identification.

C-block

The composition descriptor is modelled and implemented as a **C-block**, as a complementary block in addition to the F-, D- and S-blocks in the Computation Behaviour Composition Pattern. Each F-block has a unique C-block to describe the structural characteristics.

There are two major advantages of using dedicated C-block:

1. C-block provides sufficient and necessary structural information for run time recomposition

Listing 4.1: Metadata struct

```
typedef struct{
    int type;
    int id;
}metadata_t;
```

2. C-block can be removed after the structural composition to save memory at run time

In the C implementation, memory reserved for C-blocks could be freed and recycled if run time recomposition is not required after the system is created. This is essential for embedded platforms especially microcontroller based systems, on which memory exhaustion may lead to deadlock or unpredictable run time behaviour.

4.3 Software Plan

We propose four phases in the CPS software framework implementation:

- Phase 1: Define *C structs and types* for blocks, ports and variables involved in the composition
- Phase 2: Implement functions to *compose and validate* F-blocks
- Phase 3: Implement functions *deploy and execute* functions in F-blocks
- Phase 4: Implement functions to *create a system* using the containment and connectivity arrays

4.4 Coding Phase 1: Struct and Type Definitions

4.4.1 Metadata

We define a struct to keep the metadata of blocks, including block ID and block type as shown in Listing 4.1.

The `type` field specifies the block type, e.g. functions, capabilities or tasks for F-blocks; or data types for D-blocks. The `id` field is the global ID of the

Listing 4.2: An example of metadata structs for an F-block and a D-block

```
metadata_t md_fb_system = {
        .type = CPB_TASK_ORIENTATION_MONITORING,
        .id = FB_GID_1
};

metadata_t md_db_1 = {
        .type = DB_SENSOR_CONFIG,
        .id = DB_GID_1
};
```

Listing 4.3: F-block struct

```
typedef struct c_block c_block_t, *c_block_p;
typedef struct f_block f_block_t, *f_block_p;
typedef struct d_block d_block_t, *d_block_p;
typedef struct s_block s_block_t, *s_block_p;
typedef struct port port_t, *port_p;
struct f_block {
  c_block_p  c_blk;
  s_block_p  s_blk;
  d_block_p* d_blk;
  union{ /*either CFB, or CPB*/
      int(*func)  (f_block_p f_blk);     //CFB
      f_block_p*  f_blk;                 //CPB
  };
  port_p* ports;
};
```

block. We only use `metadata` to identify F-blocks and D-blocks in the CPS software framework, as C- and S- blocks are unique in their containing F-blocks. Listing 4.2 demonstrates the metadata of F-block `FB(1)` and D-block `DB(1)` in the orientation monitoring system presented in Section 2.11.

4.4.2 Struct of F-block

The F-block struct is defined in Listing 4.3.

Listing 4.4: C-block struct

```
struct c_block {
   metadata_t    md;
          int    nports_registered;
          int    nfbs_registered;
          int    ndbs_registered;
          int    nconns_registered;
          int    nfbs;
          int    ndbs;
          int    nconns;
          int    nports;
        char*    f_name;
         char    stat;
};
```

In C language, a `union` allows storing different types of data at the same memory location. In the struct, the union has two members: (1) a function pointer, defined for CFB; and (2) an F-block pointer array, defined for CPB.

An F-block contains a unique C-block for the structural description and several ports for data exchange. A composite F-block may also contain multiple child F-blocks and multiple D-blocks, and a unique S-block to schedule the triggering order of the child F-blocks.

The struct in Listing 4.3 formulates a complete F-block *container* that can be used to define either a computation function block (CFB) or a composite computation block (CPB).

4.4.3 Struct of C-block

C-block is the composition descriptor of an F-block. All the structural information are kept in the C-block as shown in Listing 4.4:

The C-block has all the structural knowledge of the F-block it belongs to, including the F-block type and ID in the metadata `md`, the total and current registered numbers of child F-blocks, D-blocks, ports and connections. An F-block is completely composed only when all the blocks, ports and connections are registered. The F-block completeness is examined by observing the status register `stat` to detect composition error.

Listing 4.5: D-block struct

```
struct d_block {
    metadata_t md;
    void* data;
};
```

Listing 4.6: S-block struct

```
struct s_block {
    port_p sched_list;
    int(*trigger) (s_block_p s_blk);
};
```

4.4.4 Struct of D-block

D-blocks are expected to store data of any type. In C programming, it can be realized by a simple but commonly used trick: `void` pointer and type casting.

We define the D-block struct in Listing 4.5.

Similar to the metadata in F-blocks, the metadata `md` field helps to identify block and block type. The data pointer can be cast to any desired data type pointer in practice. The `type` field in `md` helps to verify the data type before the D-block is used in computation to avoid illegal memory access.

4.4.5 Struct of S-block

The S-block struct is defined in Listing 4.6.

The S-block is connected with a dedicated scheduling list through its port `sched_list`. A triggering function is assigned to the S-block to trigger the child F-blocks in the sequence defined in the scheduling list.

4.4.6 Struct of Port

The port struct is defined in Listing 4.7.

Listing 4.7: Port struct

```
struct port {
    int port_type;
    d_block_p db;
};
```

We define `port_type` to distinguish input, output and inout ports. Multiple ports can be connected to the same data block using the D-block pointer `db`, but not the other way around. Actually, it is usually forbidden to connect multiple data sources to the same input port. For instance, in LabVIEW, connecting multiple sources to an input terminal will break the wire connections and result in an error.

In the CPS software framework, we **do not** prevent connecting multiple output ports to the same D-block. As *making connection* is technically assigning pointer of a D-block to the `db` field of a port, it is possible to connect the same D-block with multiple ports. In practice, it could be useful on embedded devices for the following reasons:

- It is safe to reuse D-block as shared memory on a single-threaded embedded system
- D-blocks can be shared to reduce memory consumption, as long as the execution order is properly scheduled

Therefore, we do not set any constraint on writing a D-block by multiple F-blocks due to the single-thread characteristic, and it is not an obstacle on multi-threading applications as long as the memory allocation is properly managed.

4.5 Coding Phase 2: Containment and Connectivity Functions

The containment and connectivity of a system is described by the AOI representation as explained in Section 2.5.1 and Section 2.5.2. The arrays are the keys to compose the structure of the computation behaviour.

4.5.1 Validating the Containment

The containment array CT is validated and analyzed by a function `cpsValidateCT` which extracts the following information:

- Number of F-blocks
- Number of child F-blocks in each CPB
- Numbers of transits and levels
- The global IDs of child F-blocks in each CPB

The function will return an error `errInvalidCT` if the containment array is invalid. For instance, $CT = \{\ 0, 1, 2, 1, 3, 3\ \}$ is an invalid containment array since the elements are not monotonically increasing.

4.5.2 Building Composite F-blocks with Connectivity

The connectivity of a CPB can be restored using its connection and port map arrays. As discussed in Section 4.4.3, connecting a port with a D-block is practically a pointer assignment to the `db` field of that port; similarly, a port mapping is a pointer assignment which passes the `db` pointer of an internal port to that of an external port.

To build up the connectivity of the entire system that usually involves multiple CPBs, we also need the following extra information:

1. Numbers of internal and external ports of each CPB
2. Type of each F-block in the system, either CPB or CFB

In the CPS software framework, we need to allocated memory for every port, no matter whether the port is connected or not. As unconnected ports (such as `IPT<4>` of `FB(1)` shown in Fig. 2.7) are not listed in the connection arrays, it is essential to specify the number of internal and external ports for precise memory allocation.

The types of F-blocks are extracted from the containment array by the `cpsValidateCT` function.

We defined a function `cpsBuildCPB` to compose CPBs. The composition is carried out in three steps:

Listing 4.8: Functions needed by `cpsBuildCPB`

```
f_block_p cpsCreateFB(...); //Creating an F-block
d_block_p cpsCreateDB(...); //Creating a D-block
s_block_p cpsCreateSB(...); //Creating an S-block
c_block_p cpsCreateCB(...); //Creating a C-block
port_p cpsCreatePort(...);  //Creating a port

int cpsCheckFbStatus(...);  //Check F-block completeness
int cpsAddFB(...);          //Adding child F-block to a CPB
int cpsAddDB(...);          //Adding D-block to a CPB
int cpsAddSB(...);          //Adding S-block to a CPB
int cpsAddPort(...);        //Adding a port to an F-block
int cpsMakeConn(...);       //Making port-DB connection
int cpsMakePortMap(...);    //Making port-port mapping
```

- Creating blocks and ports needed by a CPB
- Adding the created blocks and ports into the CPB
- Making connections and port maps

Listing 4.8 shows the low-level building functions invoked in `cpsBuildCPB` when creating a CPB.

4.5.3 Checking F-block Completeness

F-blocks are checked by `cpsCheckFbStatus` for composition completeness by examining the `stat` in its C-block, before they are added as child F-blocks in CPBs. Therefore, the CPBs should be created starting from the furthest level in the tree graph back to the root to ensure that all child F-blocks are complete.

Listing 4.9: A CPS function template

```
int cps_function(f_block_p f_blk){
  //.. Function body
  return ERR_CODE;
}
```

4.6 Coding Phase 3: Function Deployment, Scheduling and Execution

4.6.1 Function Definition

A CPS function is assigned to the function pointer in a computation function F-block (CFB). A CPS function template is defined in Listing 4.9.

The F-block pointer `f_blk` is the only argument of the CPS function, therefore it is possible to access both computational and non-computational data by de-referencing the pointers of the ports and the C-block in `f_blk`.

4.6.2 Deployment

Assigning a CPS function to a computation function block is known as *function deployment*. We define a function `cpsAssignFunc` to assign, replace or remove a function of an F-block, without influencing the structure.

4.6.3 Scheduling

Child F-blocks are triggered by the S-block contained in the same CPB. We define the schedule struct in Listing 4.10.

The schedule keeps the number of child F-blocks to be triggered in `n_child_fbs` and the child F-blocks pointers in `child_fbs`. Variable `n_child_fbs_registered` is the number of child F-block already registered during the construction, it must be equal to `n_child_fbs` to imply the completeness of the complete. The schedule can be modified at run time.

Listing 4.10: Struct for a shedule

```
typedef struct schedule schedule_t, *schedule_p;
struct schedule{
    int n_child_fbs;
    int n_child_fbs_registered;
    f_block_p* child_fbs;
};
```

4.6.4 Execution

We trigger the F-blocks using `cpsTriggerFB` function at run time. If the F-block being triggered is a CFB, the deployed function will be executed; if it is a CPB, the child F-blocks will be triggered following the sequence defined in the schedule. For example, in the system demonstrated in Fig. 2.8, triggering `FB(1)` results in the following operations:

- As indicated by the S-block, the child F-blocks are triggered in the order of `FB<1>`, `FB<2>` and `FB<3>` (or `FB(2)`, `FB(3)` and `FB(4)` respectively) according to its scheduling list
- `FB(2)` is a CFB, it is triggered to execute its assigned function
- `FB(3)` is a CPB, so its child F-blocks are triggered in the order of `FB<2>`, `FB<3>`, `FB<1>` and `FB<2>` (or `FB(6)`, `FB(7)`, `FB(5)` and `FB(6)` respectively) according to its scheduling list
- `FB(4)` is a CFB, it is triggered to execute its assigned function

`FB(1)` could be triggered multiple times to repeat the above operations. Typically, in an microcontroller-based application, `cpsTriggerFB` is called in an infinite **loop** to keep the F-blocks running.

4.7 Coding Phase 4: System Factory

We define a `cpsSystemFactory` function to automate the system creation. The substance of the automation is invoking `cpsBuildCPB` to create CPBs in the reversed order listed in the array of CPB extracted by `cpsValidateCT`. Essential checks are applied to ensure the F-block completeness during the system creation.

The `cpsSystemFactory` function deploys a `selfCheck` CPS function on every CFB to report the block type, number of input and output ports and the D-block global ID connected with each port; it also assigns a default schedule to the S-blocks in CPBs so that every child F-block can be triggered to execute the assigned `selfCheck` CPS function.

4.8 Conclusion

This chapter discussed the implementation of the Cyber-Physical Stack software framework guided by the encoded formal CPS meta model. As one of the major contributions of this thesis, the software framework realized and automated the composition of computation behaviour, and it supports run time behavioural reconfiguration and structural recomposition. Since a cyber-physical system can be described using the CPS layers and the structural model, it is straightforward to encode and create the system using the CPS software framework, in which the behavioural concerns proposed in the Composition Pattern can be implemented using the CPS function template for F-blocks.

The software framework is cross-platform and embedded-friendly. The code and functions are optimized for devices with limited memory, and they also fit non-embedded applications. In the current implementation, a C-block consists of ten `int` type, one `char` type variables and one `char` type pointer. On a 16-bit microcontroller, such as a popular ATmega328p MCU integrated on an Arduino Uno board, it occupies 23 bytes[1] in the memory. Assuming that a cyber-physical system needs 10 F-blocks for its structure, 230 bytes will be consumed by the structural descriptions, which is, 1/10 of the SRAM. We may store these data in the program memory [2] to save the precious SRAM on Atmel microcontrollers, but it is quite platform specific and is not always applicable on other platforms.

Composition descriptors are only needed in the structural composition and recomposition, and they are independent from the computation behaviour. If an F-block is not involved in run time recomposition, the memory space occupied by the C-block can be released. The motivation of memory saving was triggered by the concern of *embeddability*, which is, one of the focuses of this thesis: Device resources should be efficiently and effectively utilized. As a best practice, we advocate to explicitly decouple structure and behaviour properties as it brings in opportunities to optimize the performance and resource consumption in system design.

[1] On 16-bit systems, an integer takes two bytes, therefore the 10 `int` type variables needs 20 bytes; a pointer also occupies two bytes, and the `char` type variable needs one byte.

The core engine of the software framework currently amounts 1214 lines of code[2], it is a compact solution to model the composition of computation behaviour for autonomous self-reconfigurable systems, and it is a simple but necessary step to verify the effectiveness of the NPC4 meta model. The CPS software framework is also an easy-to-use teaching material in the software project of the Embedded Control System course. The structure and behaviour separation practically made it easier to motivate and teach software programming beginners, as an important experience learned from the education activities using the CPS software framework.

The development of the CPS software framework inspired and motivated the implementation of the composable Finite State Machine (cFSM) discussed in Chapter 3, as both CPS and cFSM meta models conform to the NPC4 meta model. The details of the cFSM software will not be discussed in the thesis, please check out the following link for the cFSM implementation: `https://gitlab.mech.kuleuven.be/u0066910/ces`. The core engine of the cFSM currently amounts 653 lines of code, it serves as a light-weight tool to model and implement state machines for coordination. A composite state can be stored in a D-block in the CPS software framework, being triggered by a state machine handler or coordinator function assigned in an F-block. The CPS software framework plus the cFSM are the two cores of the **Composable and Embeddable Software** (CES) contributed by this thesis. In Chapter 6, we will present a complete system design example using the CES running on FabLab-built, low cost setups.

[2]Calculated using the *cloc* tool in Linux

Chapter 5

Device Resources on Embedded Hardware Platforms*

This chapter explores the characteristics and advantages of programmable logic devices, in particular, the Field Programmable Gate Arrays (FPGAs) in system design. The work in this chapter is promoted by research objective 4: *Best practices for designing interfaces and algorithmic computations on embedded devices.*

In practice, FPGA logics are reprogrammable *device resources* on the devices layer (L0) in the Cyber-Physical Stack. As discussed in Chapter 2, hardware devices are interfaces connecting the control systems and the physical world, and they are also physical carriers of communication and algorithmic computations. It is therefore necessary and essential to investigate various embedded platforms to enrich the resources for the devices layer (L0).

We summarize four typical types of FPGA functions: **signal conditioning**, **signal interpretation**, **algorithmic computation** and **signal generation**. These function types are briefly explained and demonstrated using robot examples to introduce *what* can be added on the devices layer (L0) using

*This chapter is partially based on Zhang, L., Slaets, P., Bruyninckx, H. (2014). "An open embedded industrial robot hardware and software architecture applied to position control and visual servoing application", International Journal of Mechatronics and Automation, Vol. 4(1), page 63-72.

FPGAs, and to share the best practices on *how* to (re)create and (re)use the programmable logic resources in robot and cyber-physical system design.

The discussion in this chapter is on the basis of the prior experience on Xilinx FPGA devices and electronic design automation (EDA) tools: Xilinx ISE Design Suite with Embedded Development Kit (ISE & EDK), and Vivado Design Suite with High Level Synthesis (Vivado & Vivado HLS). These tools are optimized for the devices and also provide good practices in the system design process using Xilinx FPGAs.

5.1 Embedded Devices and Co-processing

ARM-based, microcontroller-based and FPGA-based embedded hardware platforms are the major **embedded devices** used in this thesis. Both microcontrollers and single-board computers are based on sequential architectures, i.e. the instructions are executed one after another using multi-function arithmetic logic units. On these devices, CPUs need to handle computations, interrupts, inputs and outputs and so on. As the workload could be extremely heavy in computation intensive applications, CPUs may suffer from additional overheads such as those brought in by an operating system. In contrast, FPGAs have dedicated programmable logic as **coprocessors** and hence are not constrained by the complexities of such overheads, in addition, one can create soft-core processors using the FPGA fabric to handle sequential programs and even operating systems.

However, implementing a soft-core CPU in FPGA fabric is typically resource consuming, and the soft processing cores are usually simpler and slower than the dedicated embedded processors such as ARM and PowerPC. FPGA System-on-Chip (SoC) technology offers a more powerful solution by integrating FPGA fabric and general purpose CPUs on a single chip, it is a compact embedded platform for cyber-physical systems with low power consumption, various IO ports and on-board peripherals, and high computation performance characteristics.

We define two major functionalities of the coprocessors in the Cyber-Physical Stack context: (1) interfacing with sensors, actuators and other devices, i.e. **interfaces**; and (2) performing algorithmic **computations**. Both functionalities are implemented and deployed on the programmable logic resources in FPGA fabric.

5.2 Programmable Logic Resources on FPGAs

There are four major types of programmable logic resources on the FPGA fabric: flip-flops, lookup tables, DSPs and Block RAMs.

1. Flip-flop

   A flip-flop (FF) is a shift register that synchronize binary logics and states. It is triggered by an input clock and it latches either true or false state.

2. Lookup table

   A lookup table (LUT) is a list of output values with respect to the input ones. Using lookup tables could significantly reduce computation time and simplify the logic circuits.

3. DSP

   DSP slices enhance the speed and efficiency in applications. They are primary options when building complicated circuitries for mathematical and digital signal processing applications.

4. Block RAM

   RAM blocks (BRAMs) are dedicated memory modules for data storage. A BRAM usually contains several kilobytes of RAM.

Typically, flip-flops and lookup tables are grouped in slices, forming up basic logic units of an FPGA known as configurable logic blocks (CLBs). The above resources are associated with each other by **programmable interconnects** for desired functionalities and behaviours. The programmable resources and programmable interconnects are surrounded by **I/O blocks** that allow the circuit to exchange data with other devices, as shown in Fig. 5.1.

The programmable logic resources are utilized to build dedicated co-processing function units. These units are usually highly reusable coprocessors and can be packaged as **intellectual property (IP) cores**. IP cores are device-independent since they describe only the behaviour of the packaged functions, the EDA tools are responsible to introduce device-specific constraints when using IP cores in different design.

To effectively use the programmable logic resources, designers must think *in hardware*, which means, variables and functions are digital signals and circuits. Since logic gates are physically interconnected with each other, the connections must be efficiently and effectively routed to avoid timing issues [59].

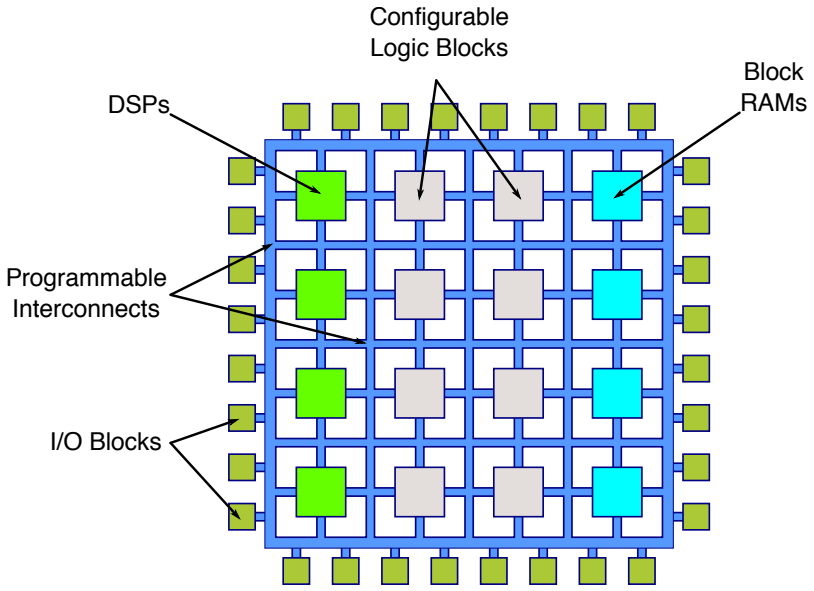

Figure 5.1: A schematic diagram of the FPGA fabric architecture. Flip-flops and lookup tables are grouped in CLBs. CLBs, DSPs and BRAMs are connected by programmable interconnects. I/O blocks are the bridges between the circuit and the outside world.

5.3 FPGA-Based Motor Control

This section shares some practical experience on interfacing an industrial robot system using the FPGA programmable resources, base on the research presented in [139].

5.3.1 An FPGA-Based Industrial Robot Arm

Current commercial industrial robot controllers [1, 73] are based on modular hardware architectures, consisting of a central processor behaving as a coordinator and various additional co-processors performing low level motor control. These industrial architectures are designed to ensure robot reliability and maintainability. The commercial industrial software is usually integrated in the hardware platform and focuses on position based motion control applications; in addition, lack of extendibility with alternative sensors or axis makes it difficult,

or even impossible to extend the functionality of the robot. To address the require of complicated motion control, as well as implementing control algorithm, open source robot control software alternatives [13, 44, 47] are being developed, maintained and widely used.

In this section, an open embedded hardware and software architecture is presented. This architecture can be easily extended to control a general multi-axis robot or a mechatronic system, as long as the motors are equipped with incremental position decoders and driven by pulse width modulation (PWM) signals. In addition, this architecture brought in the inspiration of composing hardware and software **components**[1] on a single platform, which contributed to the preliminary concept of the Cyber-Physical Stack.

In the setup shown in Fig. 5.2, a Performer MK-2 industrial robot arm is driven by a Xilinx Virtex-II pro FPGA SoC. The motor controller IP core was originally implemented using Very high speed integrated circuits Hardware Description Language (VHDL). One of the PowerPC processors runs a real-time Linux and OROCOS [13]. The software system, consisting of the Linux kernel and configuration file of the programmable logics, is stored in an bootable ACE-file and is uploaded to the FPGA SoC using a compact flash (CF) card at power up. The Linux root file system can be stored on the same card, or alternatively on a remote PC using a network file system (NFS). The PowerPC is connected to the FPGA via a processor local bus (PLB). Commercial MD10C NMOS H-bridged motor drivers, designed by Cytron Technologies, are interfaced with the FPGA SoC to drive the DC motors in this setup. The MD10C supports up to 10 kHz PWM signal for DC motor control.

We extract the following device resources besides CPU and memory from Fig. 5.2: five motor controllers, an Ethernet Controller, a UART, a SysACE Compact Flash driver and an image processing unit with camera driver. Each of them is an individual coprocessor in the system, they are created on the FPGA fabric using the programmable logic resources and run in parallel. As device resources in the Cyber-Physical Stack, these coprocessors are initiated and (re)configured by corresponding driving functions at run time.

Fig. 5.3 depicts the conceptual model of the same system shown in Fig. 5.2 in the Cyber-Physical Stack context. The previously mentioned device resources are reformulated in the devices layer (L0) in a Cyber-Physical Stack. The `visual servoing` task is realized by two capabilities: `object tracking` and `Cartesian space position control`. To keep the conceptual model simple, we also hide the functions related with human interface (via

[1] The term **component** used in this section refers to a functional entity deployed either on hardware or in software, which is comparable with the blocks in the Cyber-Physical Stack meta model.

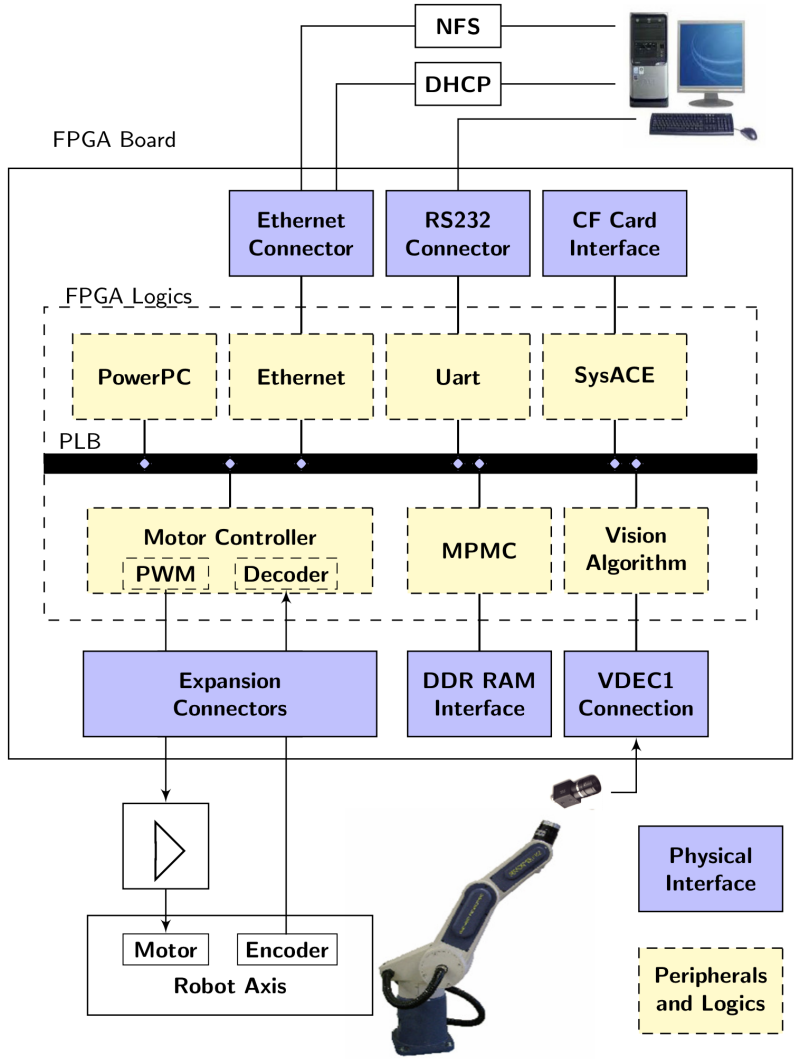

Figure 5.2: The embedded robot control architecture consists of four parts: an industrial robot, a power control bridge, a terminal interface and an FPGA SoC board.

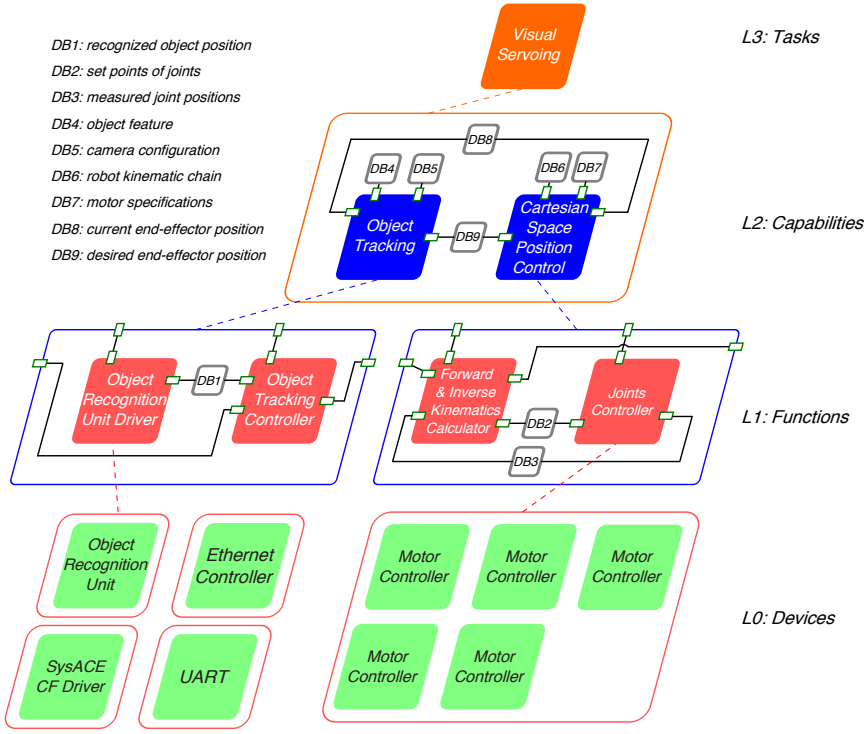

Figure 5.3: The conceptual model of an FPGA-based visual servoing system in the Cyber-Physical Stack context. The IP cores are device resources on the devices layer (L0) in the CPS meta model.

`UART` and `Ethernet Controller`) and file system (`SysACE CF Driver`) in the conceptual model. The `Object Recognition Unit` implemented using FPGA programmable logic resources is driven by the `object recognition unit driver` function, and the five motor controller IP cores are driven by a `joints controller` function. System-specific knowledge such as **motor specifications** and **kinematic chain** of the robot must be provided to configure the `Cartesian space position control` capability; and **object features** and **camera configurations** are needed for the configuration of `object tracking` capability. The set points in Cartesian space of the end-effector of the robot arm is determined by `object tracking`, based on the current position of the target object in the image, as well as the current position of the end-effector in Cartesian space.

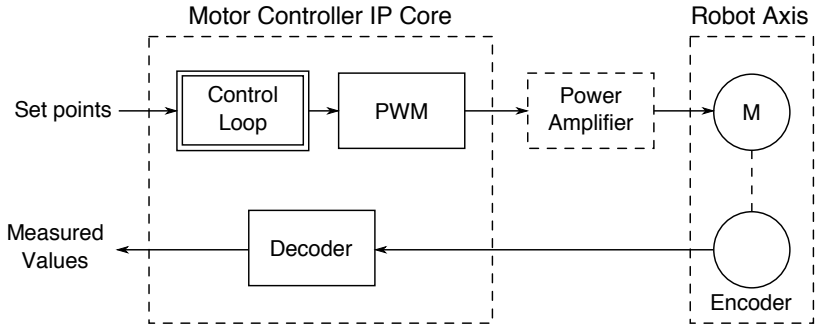

Figure 5.4: The complete motor control schematic.

It is not the intention for this section to dive into the implementation details of the object recognition unit driver. Instead, we will focus on the FPGA design of the highly reusable **motor controller IP core** that contributes to the functionality of the robot arm and an FPGA-based mobile platform discussed in Section 5.4.

5.3.2 Designing the Motor Controller IP Core

Fig. 5.4 illustrate the motor control schematic diagram. The motor controller IP core consists of a quadrature decoder, a PWM signal generator and a control loop, which covers four typical function types of FPGA: (1) signal conditioning, (2) signal interpretation, (3) algorithmic computation and (4) signal generation. We will introduce the four typical applications using the quadrature decoder and the PWM signal generator as examples.

The Performer-MK2 has five joints, each joint is equipped with a quadrature optical encoder. The two quadrature output signals `A` and `B` are 90 degrees phase different. By sampling `A`, `B` and an additional index signal `I` that generates a pulse in every revolution, relative rotational position, velocity and acceleration can be determined and calculated. Fig. 5.5 gives a schematic of the decoder interface, which consists of seven computation units: three input noise filters, a quadrature decoder, a position calculator, a velocity calculator and an acceleration calculator.

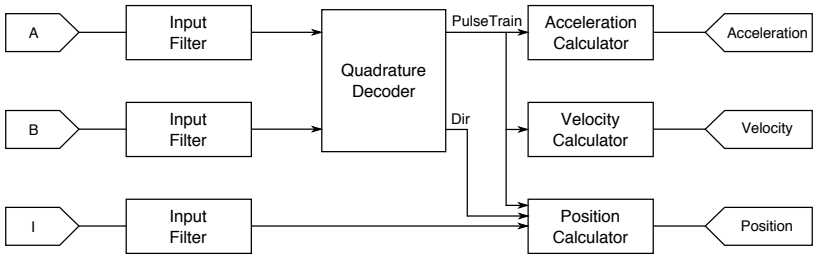

Figure 5.5: A decoder interface consists of seven computation components: three input noise filters, a quadrature decoder, a position calculator, a velocity calculator and an acceleration calculator.

1. Signal Conditioning: Input Filters

The filters perform digital signal conditioning to eliminate high frequency noise present in the input quadrature encoding signals. Every digital input is passed through four D-type flip-flops (D-FF) in four consecutive clock cycles. The output value only follows the input when the latter three D-FF outputs coincide. The filter operates properly when the clock frequency is at least two times faster than the decoding signal capturing frequency, which is approximately 70 kHz in the current setup. The filter, however, introduces a delay of four clock cycle, but this delay can be ignored by a relatively high sampling frequency e.g. 10 MHz.

2. Signal Interpretation: Quadrature Decoder

The filtered digital signals `A` and `B` are decoded and interpreted as a pulse train signal `PulseTrain` and a rotating direction signal `Dir` by the quadrature decoder. The two decoded signals are used to determine the current position, velocity and acceleration of a motor or a robot axis. Based on [51], the state diagram shown in Fig. 5.6 is implemented in VHDL.

The logic state machine starts with the `Idle` state, followed by the *Waiting* state in which the state machine waits for valid input signals `A` and `B`. The next state is determined by both `A` and `B`, in a combination of `00`, `01`, `11` and `10`. The `PulseTrain` and `Dir` signals are generated with respect to the change of `A` and `B`.

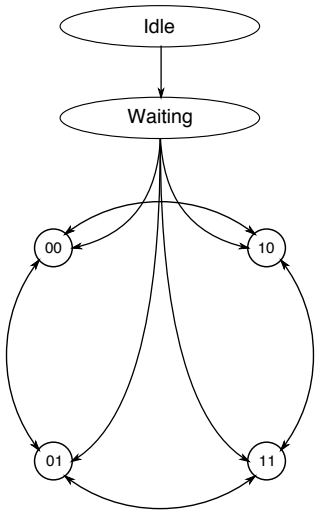

Figure 5.6: The quadrature decoder state machine implemented in VHDL.

3. Computation: Position Calculator

The position calculator calculates the current position based on the number of complete revolution and rotated angular position, which are determined by a revolution counter and an angle counter. The position is calculated by:

$$Position = \frac{(rev \cdot m + angle) \cdot \theta}{n} \tag{5.1}$$

where m, θ and n are robot axis and motor dependent parameters: m denotes the number of pulses for each revolution, θ denotes the angle corresponding to a state shift of the quadrature decoder, and n corresponds to the transmission ratio between the motor and the axis [43].

The velocity calculator, acceleration calculator and the control loop are also computations implemented using the programmable logic resources. However, as it is not the intention to discuss control algorithms and computation details, we only focus on the resource consumption issues concerning with these **computations** in later sections.

4. Signal Generation: PWM Signal Generator

The output of the control loop is converted to a 12-bit signed integer plus an offset to produce a lock anti-phase PWM signal. The motor stays standstill at 50% PWM duty cycle, and it rotates clockwise or counter-clockwise when the duty cycle is below or above 50%.

The PWM signal generator implemented in the motor controller is capable to generate PWM signals up to 25kHz. However, the power amplifier Cytron MD10C limited this frequency at 10kHz, resulting in an acoustic noise in the operation.

5.3.3 Resource Utilization

The availability of programmable logic resources is an inevitable constraint brought in by the FPGA devices. The Virtex-II Pro board has 13696 configurable logic blocks (CLBs) [134], which is equivalent to 30816 logic cells[2]. It has 136 18kb-BRAMs and does not have dedicated DSP cells.

As all such resources are explicitly modelled in the EDA tool, we can review the resource consumption of the *implementation* before it is *deployed* on the FPGA platform. In the design of the proposed hardware and software architecture, each motor controller IP core consumes 22% of the total CLBs, implying that the FPGA would not be capable to drive a 5-axis robot like the Performer-MK2 since the required amount of resources has exceeded 100% of the board.

The resource exhaustion problem was addressed by replacing floating-point representation with *fixed-point binary* [93]. The precision is actually determined by the resolution of the quadrature encoder for position calculation. On the Performer-MK2 robot, a full axis revolution counts 2000 pulses, the equivalent resolution is therefore $2\pi/2000 = 0.00314159(rad/pulse)$. This decimal fraction can be appropriately represented by a 11-bit fixed-point binary fraction, in which the least significant bit represents 0.000488281 and is approximately 1/10 of 0.00314159. As a consequence, the binary fixed-point approximation reduced the resource consumption of a motor controller IP core to 10% of the overall available resources of the Virtex-II Pro FPGA, while retaining an adequate precision in the computation.

[2]Logic cell is a convenient unit of abstraction for quantifying resources. On the Virtex-II Pro FPGA used in this setup, each logic cell is made up of a 4-input look-up table (LUT) and a flip-flop. *Equivalent Logic Cells* equals *Total CLBs* x 8 (logic cells per CLB) x 1.125 effectiveness [135]. Therefore, on the Virtex-II Pro FPGA, there are 30816 equivalent FFs and 30816 equivalent LUTs.

To minimize the precision loss and to avoid overflow, we must assign sufficient bits for both fractional and integer parts and define the range of the fixed-point binary in advance. Fixed-point arithmetic requires less complicated hardware than that for emulating floating-point, thereby reducing the consumption of the FPGA resources. Moreover, the reduction of hardware complexity also cuts off the execution time as floating-point takes more clock cycles in the circuit.

5.3.4 Choice of Formal Languages for Algorithmic Computations

The motor controller IP core was programmed using VHDL. As a formal language for describing the structure and behaviour of digital circuits, it is intuitive to design concurrent system using VHDL since parallel processing is explicitly supported. Moreover, hardware description languages (HDLs) also support gate level abstraction, it is therefore straightforward to model and implement digital signal conditioning and generation functions, in which the inputs and outputs are logic signals `'0'` and `'1'`.

Most of the design efforts were spent on the computation functions such as the control loop and the position calculator. First of all, it is difficult to simulate a VHDL design when the computation is relatively complex, especially when fixed-point values are involved in the computation, frequent type conversions may bring in extra complexity, and therefore increasing the debugging difficulty. Secondly, it is rather time consuming when describing mathematical computation in VHDL. The motor controller IP core was implemented with approximately 1580 lines of hand-typed code, in which 689 lines for the algorithmic computations, and 653 lines for signal conditioning, interpretation and generation. Lastly, programmers need to "think in hardware", i.e., they are expected to have knowledge of hardware circuits and the availability of programmable logic resources before developing algorithmic computations, as an additional constraint brought in by HDLs.

To address the above issues in designing complex algorithmic computations, aiming at reducing the development time as well as the overall design effort, we suggest to use EDA tools such as Vivado HLS to develop the computation intensive algorithmic coprocessors. These tools usually hide the hardware details and allow designing algorithms using high-level and general purpose programming languages such as C and C++ [136], making it easier to model and implement algorithmic computations. Following appropriate coding styles, C/C++ code would be synthesizable and could be packaged as reusable IP cores. The work of Cong et al. [21] discussed the power of using HLS in shortening the time-to-market of an FPGA system design. HLS-facilitated FPGA designs

have covered different domains including 4G wireless system [48], aerospace application [101], image processing [29, 112, 30] and cosmology data analysis [63].

5.3.5 Redesigning the Motor Controller IP Core

In the revision of the motor controller IP core, we explicitly separate the four function types as proposed at the beginning of this chapter, and reimplemented the **algorithmic computations** including the position, velocity and acceleration calculators, as well as the control loop in Vivado HLS using C/C++.

The reimplementation of the above calculators amounts to in total 32 lines of C code, while a simple PID controller costs 33 lines. The C codes are converted to VHDL and packaged as individual IP cores, and are used to compose the redesigned motor controller IP core afterwards.

Comparing with the amount of code of that in VHDL (689 lines), using Vivado HLS and developing algorithms in C/C++ saved tremendous time on coding and debugging. Moreover, Vivado HLS provides its own math library which contains the most commonly used functions including fixed-point data type, resulting in smaller and faster hardware for a small loss of accuracy. By defining appropriate marcos in the C code, one can easily switch between fixed-point and floating-point data types. As the same C/C++ implementation can be deployed on the embedded ARM processors instead of the FPGA programmable logics, the same computation actually fits on both the devices layer (L0) and the functions layer (L1) in the CPS meta model. This is a flexible feature when designing robot and cyber-physical systems using FPGA SoCs in the CPS context.

5.4 Reusing the Motor Controller IP Core in Educational Robot Design: An FPGA-Based Mobile Robot Platform

The redesigned motor controller IP core is reused on an educational robot, an FPGA-based Mecanum wheeled [31] mobile platform as shown in Fig. 5.7.

The mobile platform consists of three stacked parts in Fig. 5.7: a Zynq-7000 Series *ZedBoard FPGA SoC* on the top, an *electronics container* containing sensors and motor drivers in the middle, and an *aluminium chassis* mounted

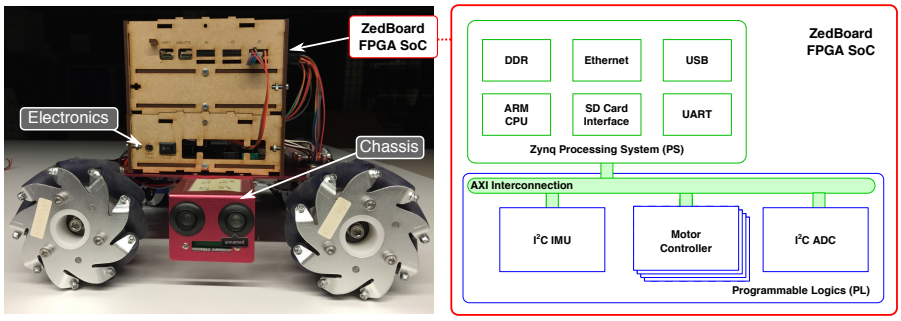

Figure 5.7: The redesigned motor controller IP core has been used on a ZedBoard FPGA SoC driven mobile platform, the model of deployment is depicted in the right-hand side in the figure.

with four individual DC motor driven Mecanum wheels at the bottom. Each of the DC motor is equipped with an optical quadrature encoder, hence it is straightforward to reuse the motor controller IP cores.

The Zynq Series FPGA boards integrated various peripherals and interfaces such as I^2C, SPI, UART, USB and Ethernet together with the dual-core ARM processors, known as the **ZYNQ Processing System (PS)**. The PS is practically an ARM-based single-board computer, running a light-weight embedded Linux in this setup. IP cores built on the FPGA fabric, or the **programmable logic (PL)**. Both PS and PL are connected to the **Advanced eXtensible Interface (AXI)** bus.

The conceptual model of the mobile platform is demonstrated in Fig. 5.8. Similar to the conceptual model of the FPGA-based visual servoing system introduced in Section 5.3.1, we hide functions for human interface and file system operation for simplicity, but focus on the **reusability** of the `motor controller IP core` device resource and the `Cartesian space position control` capability. The `joints controller` function practically drives the four `motor controllers`, with system-specific motor specifications and kinematic chain. In the conceptual model we introduce a `formation path planning` capability, composed by a `communication mediator` function and a `trajectory generator` function. These functions are necessary for tasks like `moving while keeping the formation`, therefore the mobile platform is able to joint the formation system discussed in Section 2.7.3 as a sub-system. The meaning of a message is interpreted and composed by the `communication mediator` using specific communication protocols as configuration.

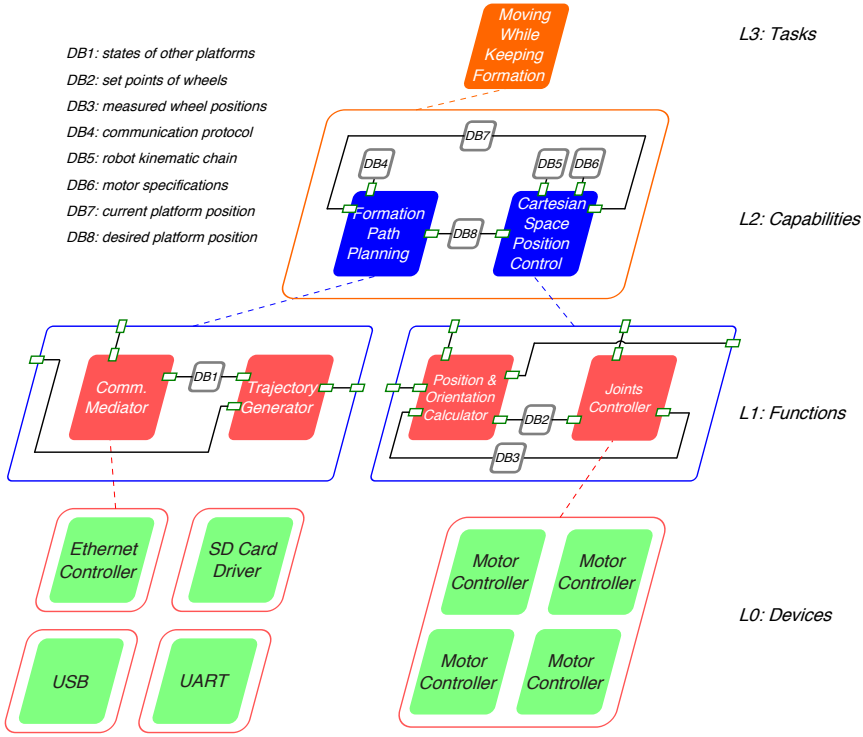

Figure 5.8: The conceptual model of an FPGA-based mobile platform in the Cyber-Physical Stack context. The IP cores used in the mobile platform as resources are formulated on the devices layer (L0).

5.5 Best Practices

In this section, we will discuss the lessons learned in the design process of FPGA SoC based systems.

Lesson 1: Identify the Four Function Types

Most of the FPGA applications could be composed by the proposed four types of FPGA functions: **signal conditioning**, **signal interpretation**, **algorithmic computation** and **signal generation**. A clear *decomposition* of the above

functions would help creating system-independent IP cores and therefore would help to *recompose* them in different applications.

HDLs are the first choices for digital signal modelling. It is intuitive to implement interfacing functions using HDLs for signal sampling, conditioning, reconstruction and generation, however, HDLs are not time-efficient in algorithmic computation design. We suggest using HLS or other EDA tools to create complex algorithmic computations to reduce development time and effort. Extensive libraries are being developed and launched for arbitrary precision data types, video and image processing, linear algebra and so on, making HLS more efficient and powerful in computation intensive IP core design.

Lesson 2: Be Critical on Resource Consumption

Resource exhaustion may prevent the deployment of the implemented IP cores as discussed in section 5.3.3. Three commonly used techniques to reduce resource consumption on FPGA-based system are listed below, in the premise of affordable accuracy loss.

1. **Using fixed-point instead of floating-point**

   Using fixed-point data type could significantly reduce the resource consumption and latency [137]. It is essential to keep adequate accuracy by using sufficient fractional bits, and it is always necessary to verify the ranges in advance to avoid overflow and precision loss.

2. **Using appropriate polynomial approximation for complex mathematical functions**

   Some mathematical functions may consume considerable resources. A 32-bit single precision exponential function $f(x) = exp(x)$ uses 11% DSP, 1.6% FF and 5.6% LUT on a Zynq-7020 device. This function can be realized by a polynomial approximation [58] and fixed-point representation, if the independent variable x varies in a fixed domain.

   For instance, $f(x) = exp(x)$ can be approximated by a quadratic function $f(x) = 0.8395x^2 + 0.8517x + 1.0126$ when $x \in (0.1, 0.9)$, with a maximum deviation of 0.47%. Instead of 32-bit single precision, we use a 32-bit signed, fixed point binary representation[3] in the function. The resource

[3] In which one bit for the sign, three bits for the whole number part and 28 bits for the fractional part.

consumption can be reduced to 4.55% DSP, 0.31% FF and 0.47% LUT, with almost doubled computation speed[4].

3. **Implementing image downsizing to reduce the consumption of computation resources and memory**

   Image downsizing could be used in some image processing applications if the precision loss is not significantly connected with the desired capability, in order to reduce the consumption of block RAMs and the latency of computation.

We carried out an experiment in the robot lab as a student project to evaluate the effectiveness of the above approximation techniques using a background subtraction algorithm [124, 96] in Vivado HLS. The result is demonstrated by Fig. 5.9 and Table 5.1.

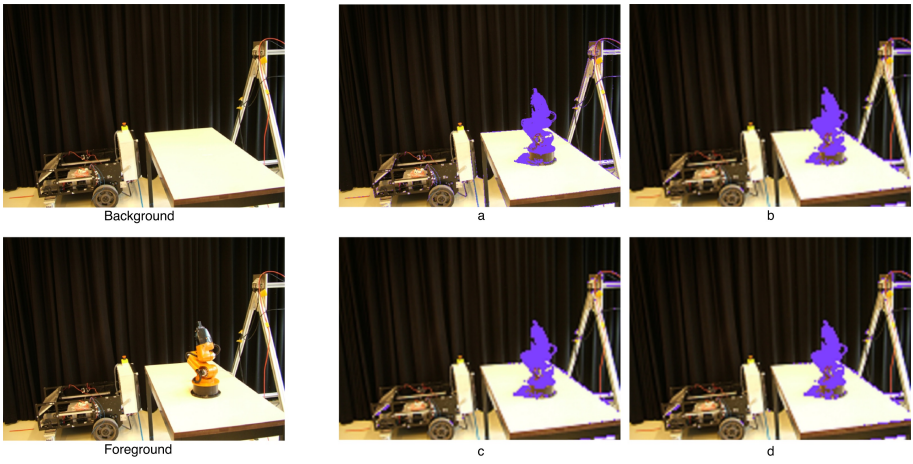

Figure 5.9: A background subtraction algorithm is implemented on a ZedBoard with different approximations. The resource consumption result is shown in Table 5.1.

As shown in Fig. 5.9, the robot arm is recognized as a foreground object marked with purple pixels with different combination of the above techniques. Downscaling the image with a factor of four resulted in tremendous saving of BRAM and significantly shortened the processing delay. Next, we replaced the non-linear functions in the algorithm with piecewise linear approximations, and

[4]Tested in Vivado HLS 2014.3. The exponential function $f(x) = exp(x)$ in single precision takes 23 clock cycles, while the polynomial approximation with fixed-point representation takes 12 clock cycles.

Table 5.1: Estimated resource utilization of background subtraction algorithm depicted in Fig. 5.9.

No.	Resolution	Precision	Math Simp.	BRAM	DSP	FF	LUT	Delay (ms)
a	640x480	Float	No	456	53	12386	19765	1157
b	160x120	Float	No	48	53	12366	19634	73
c	160x120	Float	Yes	48	27	6441	12726	40
d	160x120	Fixed	Yes	32	16	1719	3375	19

then replaced floating-point with fixed-point in the algorithm, the background subtraction result is still acceptable with a further reduced resource consumption and computation delay, without significantly compromising the algorithm capability.

Lesson 3: Apply Appropriate Design Constraints

In Xilinx Vivado and Vivado HLS, algorithmic computations can be optimized by adding proper **directives** as instructions in the design to achieve desired outcome on hardware performance, resource and power consumption [136, 118]. These directives are actually constraints as pre-configuration of hardware at compilation to override the defaults, and they usually bring in resource-performance trade-offs. For example, `set_directive_unroll` transforms a loop by creating multiple copies of the loop body so that it can be executed in parallel to accelerate the computation, however, unrolling a loop involves more programmable logic resources in return. It is worth investigating which parts are crucial to unroll in a design, especially when the resource utilization is reaching the limit.

5.6 Conclusion

This chapter explores the characteristics and advantages of using FPGAs in robot and cyber-physical system design. In the Cyber-Physical Stack context, IP cores on FPGAs are device resources that can be running in parallel. This is a very distinctive feature among the various embedded platforms as common single-board computers and microcontrollers are sequential.

We advocate an explicitly separation the four major function types when designing IP cores or coprocessors in order to effectively decompose and recompose the desired functionalities, and use different formal programming languages and EDA tools to accelerate the prototype development. As an

example, we presented a motor controller IP core that was originally implemented in VHDL for industrial robot arm control. By separating the four major function types, we reimplemented the algorithmic computations in Vivado HLS using C/C++ and afterwards repackaged and reused the IP core on an educational mobile robot platform.

Different from software programming like C or Java, we must be strict on resource consumptions in FPGA designs as computation resources are limited. We propose three commonly used techniques to reduce the resource utilization, in the premise of affordable accuracy loss. We also suggest using optimization constraints provided by EDA tools properly to sufficiently exploit the computation performance of FPGAs.

The FPGA SoC architecture seamlessly matches the Cyber-Physical Stack meta model, in which hardware devices are explicitly separated and modelled as a particular layer. Moreover, as intensive computations are dispatched on FPGA programmable logic resources, the embedded ARM processors are relieved so that the Cyber-Physical Stack software framework could focus on coordination, communication and monitoring.

Chapter 6

Prototyping: From Concept to Deployment

This chapter presents a complete system design process for a system-of-systems. This effort is motivated by research objective 5: *Education: Using the developed Cyber-Physical Stack meta model and software framework to demonstrate, introduce and teach system-of-systems design.*

We will use the meta models, design patterns and software discussed and developed in Chapter 2, Chapter 3 and Chapter 4 to model and implement systems, following the design phases proposed in Section 1.6.1.

The system-of-systems design process consists of several steps below:

1. Conceptual modelling

   Identify the desired tasks, system-specific capabilities, functions and available devices, tile them as blocks in the four layers in the Cyber-Physical Stack and sketch the connections between these blocks.

2. Event loop modelling

   Model the event loop using the concerns of the Composition Pattern, and add the missing concerns (if any) into the conceptual model.

3. Formal structural modelling

   Flatten the conceptual model as a formal structural model and apply necessary optimizations to improve the readability, embeddability and run time efficiency.

4. Computer readable AOI representation formulation

   Formalize the modelled structure of computation behaviour using the AOI representation.

5. Life-Cycle State Machine Modelling

   Create the Life-Cycle State Machine model for coordination.

6. Implementation

   Implement the structure of computation behaviour and the behavioural functions.

7. Deployment

   Add hardware-specific code to glue the implemented system with selected devices and platforms.

The above steps formulates a **system design workflow**. This workflow is generic and is applicable to any system design process using the Cyber-Physical Stack.

The ambition of this chapter is to verify the validity and feasibility of the presented design approach, meta model and software framework, and to provide a concrete example of designing and building low cost educational setups using the developed software, popular and easy-to-learn embedded hardware, and the digital fabrication equipments.

6.1 Concept

The system-of-systems concept is demonstrated by Fig. 6.1. It consists of three sub-systems: a central coordinating *computer* and two microcontroller-based *gadgets*.

It is not the intention to discuss how the central coordination is modelled and implemented on the computer system, as the coordination job for the computer is rather simple with human interaction. Instead, we are more interested in how the microcontroller-based gadgets are conceptualized, modelled, implemented and deployed, as one of the major contributions of this thesis.

The gadgets and the computer are able to send and receive messages in the communication network. Messages are visible to all systems in the network.

The computer is a system-wise coordinator coordinating the system behaviour and initiate autonomous self-reconfiguration and run time rescheduling on the

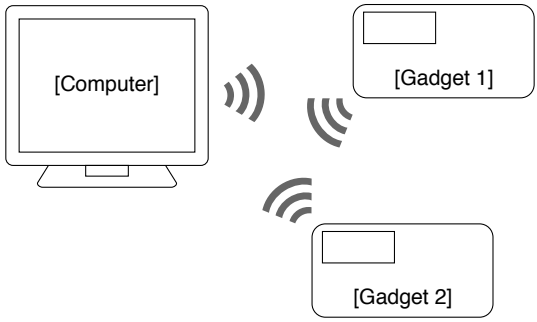

Figure 6.1: This figure demonstrates the target system-of-systems that consists of three systems: a central coordination computer for system-wise coordination, and two microcontroller-based gadgets. The three systems are in the same communication network.

gadgets. Each of the gadgets is equipped with an accelerometer, an LED ring, an LCD and a wireless communication module. A gadget can be operating in two modes (semantically, the term *mode* is a synonym for "state" of a *sub-system* in a *system-of-systems*), either in standalone mode or in peer-to-peer mode:

- Standalone mode

  The gadget detects its own orientation, and generate a corresponding colour pattern for the LED ring with respect to the orientation.

- Peer-to-peer mode

  A `sender` gadget and a `receiver` gadget are involved in peer-to-peer mode. The `sender` sends its orientation together with the `receiver`'s system ID to the communication network. The LED ring on the `receiver` demonstrates the colour pattern translated from the `sender`'s orientation. The accelerometer on the `receiver` is disabled in this mode. If the `receiver` gadget does not receive any valid message within 20 seconds, it will to standalone mode automatically.

Fig. 6.2 demonstrates an assembled microcontroller-based gadget. The mechanical structure is made in FabLab using the laser cutting machine.

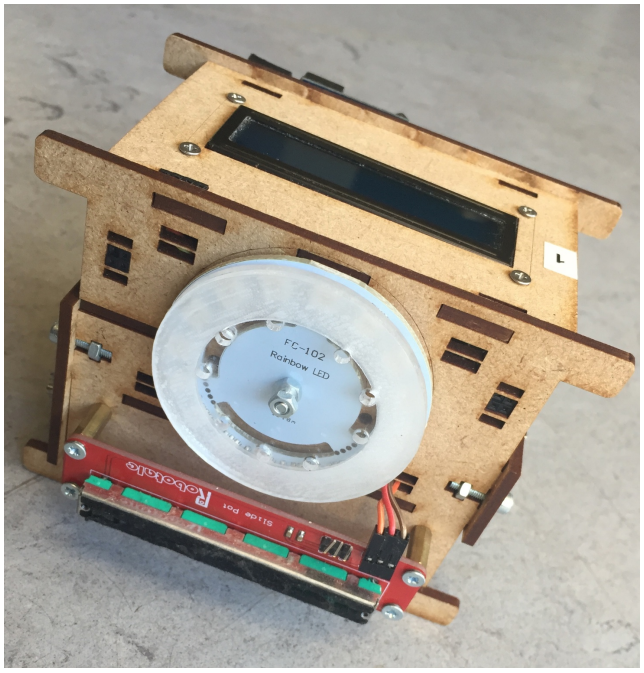

Figure 6.2: An assembled microcontroller-based gadget.

6.2 Conceptual Modelling Using Cyber-Physical Stack

The conceptual model of the embedded gadgets is formulated by the Cyber-Physical Stack meta model, as illustrated in Fig. 6.3.

6.2.1 Task

The task of a gadget is to show a colour pattern with respect to, either the orientation of the gadget itself, or that of another gadget. We define `Local&Remote LED Control` as the task, as shown in the tasks layer (L3) in Fig. 6.3.

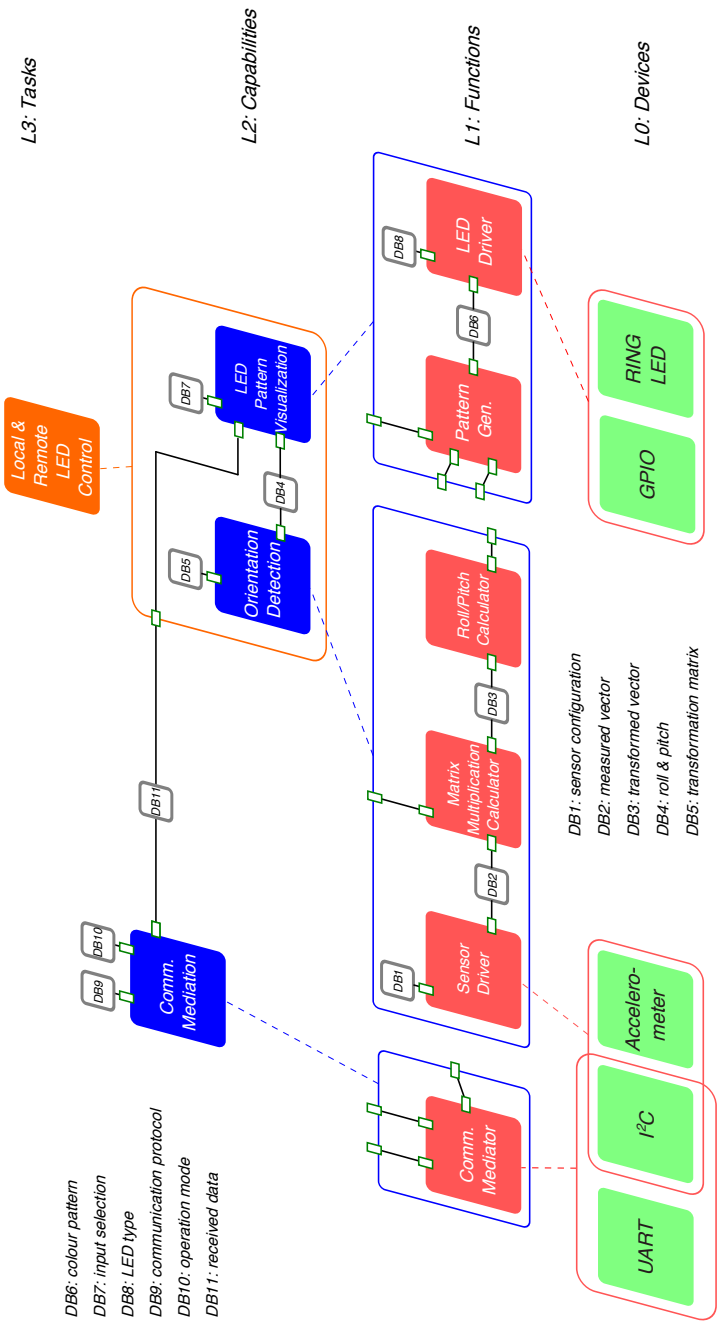

Figure 6.3: The conceptual model of the gadgets is formulated by the Cyber-Physical Stack meta model.

6.2.2 Capabilities and Functions

To compose the desired gadget task, we need three capabilities: `communication mediation`, `orientation detection` and `LED pattern visualization`, as implied in the capabilities layer (L2) in Fig. 6.3.

Communication Mediation

The `communication mediation`[1] capability is contributed by a `communication mediator` function that interprets incoming messages and composes outgoing messages, using a particular communication protocol. The `communication mediator` function is responsible for monitoring the communication device activities to send and receive messages. In peer-to-peer mode, the `sender`'s `communication mediator` function receives orientation query requests and it composes and sends messages including the orientation data and the `receiver`'s ID; the `receiver`'s `communication mediator` function composes data query requests for a `sender` gadget and it receives and interprets messages from the `sender`.

The current operation mode is stored in a dedicated D-block as a configuration of the `communication mediation` capability. If unexpected communication loss between the two gadgets lasts for longer than 20 seconds, the `communication mediator` will raise an event for the coordinator to trigger a **self-reconfiguration** to return to standalone mode.

We integrate a `logging` function in the `communication mediator` in this example, so that it has access to all the D-blocks[2] and therefore is able to compose messages with local data, which is essential for a `sender` gadget in peer-to-peer mode.

Orientation Detection

The `orientation detection` capability has been discussed in Section 2.6, as depicted in Fig. 2.9. It is **reused** in the design of the microcontroller-based gadgets as shown in Fig. 6.3. This capability detects the orientation of the gadget, and stores the measured roll and pitch in a data block for other capabilities.

[1] The mediator pattern has been briefly introduced in Section 1.6.1

[2] Logging functions are allowed to read all D- and S-blocks in the Cyber-Physical Stack, as discussed in Section 2.6.4.

LED Pattern Visualization

The `LED pattern visualization` capability is composed by two functions: (1) an `LED driver` driving the ring LED; and (2) a `pattern generator` generating colour patterns for the LED driver.

The `pattern generator` can be connected with two different sets of inputs, but only one of them is chosen for pattern generation according to its configuration at run time.

The pattern generation function is formulated by Eq. 6.1, in which ϕ and θ denote two independent input variables (pitch and roll of a gadget here), P denotes the generated pattern, and c is a binary selector determining which input is used in the pattern generation.

$$P = \begin{cases} f(\phi_1, \theta_1), (c = 0) \\ f(\phi_2, \theta_2), (c = 1) \end{cases} \tag{6.1}$$

The generated pattern P is visualized by the `LED driver`. The `LED driver` should be configured with *device-specific knowledge* (number of LEDs on the LED ring in this example).

6.2.3 Devices

The devices and interfaces used in the gadgets are listed in the devices layer (L0) in Fig. 6.3. To be more specific, the following electronic devices are integrated in a gadget:

1. A ring LED with one-wire interface

   The ring LED consists of twelve WS2812N[3] LEDs driven by a specific one-wire protocol through a GPIO pin.

2. An inertial measurement unit (IMU) with I^2C interface

   The I^2C interfaced IMU mounted in the gadget integrated an accelerometer, a gyroscope and a magnetometer. Only the accelerometer is used to determine the gadget orientation.

3. An LCD1602 with I^2C interface

[3] An individually-addressable RGB LED. Internally it includes intelligent digital port data latch and signal reshaping amplification circuit, and can be controlled and cascaded by a single line.

Listing 6.1: The event loop for the microcontroller-based gadgets in standalone mode.

```
When the system is triggered
do event_loop{
   communication_mediator()
   coordinator()
   configurator()
   scheduling::led_control(){
     scheduling::orientation_detection(){
       computation::data_acquisition()
       computation::matrix_multiplication()
       computation::roll_pitch_calculation()
     }
     scheduling::LED_pattern_visualization(){
       computation::pattern_generation()
       computation::LED_driver()
     }
   }
}
```

The LCD1602 display attached to the gadget is also interfaced with the I^2C bus. It is used as a logging device.

4. An A7105 with UART interface

   The inter-system wireless communication is realized by UART-based A7105 wireless 2.4GHz transceivers.

6.3 Modelling the Event Loop

The computation behaviour for the desired task is executed using an event loop as shown in Listing 6.1.

In order to participate as a sub-system in a complex system, the `communication mediation` capability (essentially the `communication mediator` function) must be **extended** to extract events from the incoming messages. Consequently a dedicated D-block is needed to store the received events.

A `coordinator` is needed in the event loop after the communication process. The core of the `coordinator` includes a state machine handler which operates

the Life-Cycle State Machine as a composite state held in a D-block, and an event handler that processes and raises new events (e.g. an event to trigger the reconfiguration) at run time.

If reconfiguration is required, the `coordinator` will trigger a `configurator` to modify the data stored in corresponding D-blocks, and if necessary, to change the schedules in S-blocks at run time. The `configurator` has access to **all** D-blocks and S-blocks, and it may have dedicated D-blocks to store preset configuration data.

The scheduled computation for the task `LED control` is composed by the other two *system-dependent* capabilities, `orientation detection` and `LED pattern visualization`. The event loop shown in Listing 6.1 satisfies the desired functionalities of the gadgets; however, in different operation mode, the scheduled computations are different:

- In standalone mode, all the functions are scheduled for execution in the loop at shown in Listing 6.1.
- In peer-to-peer mode, the operation of a `sender` gadget also needs all the scheduled computations, but the operation of a `receiver` gadget skips the `orientation detection` capability in the schedule as local sensing is not needed, as shown in Listing 6.2.

The event loop helps decoupling behavioural concerns and separating coordination and computation. For instance, if the LCSM is in `capability configuration` state, the scheduled computation for `LED control` will be skipped to avoid unexpected behaviour during the configuration process, as shown in Listing 6.3.

Fig. 6.4 depicts the extended conceptual model of the microcontroller-based gadgets with additional concerns introduced by the event loop. The extended and additional concern functions are highlighted by purple strokes: a `configurator`, a `coordinator`, and an *enhanced* `communication mediator`. It is worth noting that although the `configurator` has access to all data and schedules in the system, we do **not** explicitly connect the `configurator` with every D-block to retain the readability as suggested in Section 2.6.4. The same consideration applies to the `communication mediator`, in which a `logging` function is embedded.

Listing 6.2: The event loop for a receiver gadget in peer-to-peer mode.

```
When the system is triggered
do event_loop{
    communication_mediator()
    coordinator()
    configurator()
    scheduling::led_control(){
      //scheduling::orientation_detection(){
      //  computation::data_acquisition()
      //  computation::matrix_multiplication()
      //  computation::roll_pitch_calculation()
      //}
      scheduling::LED_pattern_visualization(){
        computation::pattern_generation()
        computation::LED_driver()
      }
    }
}
```

Listing 6.3: The event loop when a gadget in `capability configuration` state.

```
When the system is triggered
do event_loop{
    communication_mediator()
    coordinator()
    configurator()
    //scheduling::led_control(){
    //  scheduling::orientation_detection(){
    //    computation::data_acquisition()
    //    computation::matrix_multiplication()
    //    computation::roll_pitch_calculation()
    //  }
    //  scheduling::LED_pattern_visualization(){
    //    computation::pattern_generation()
    //    computation::LED_driver()
    //  }
    //}
}
```

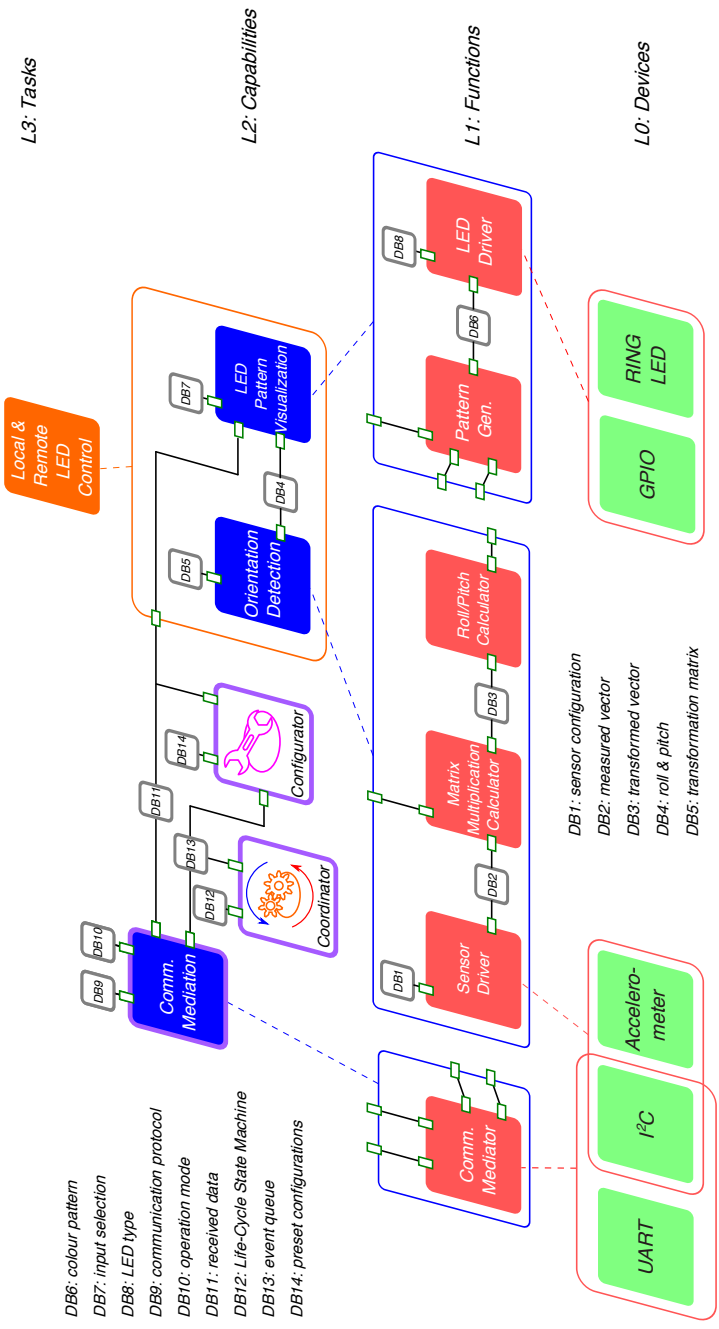

Figure 6.4: The conceptual model of the gadgets is extended with additional concern functions for the event loop.

6.4 Formalizing the Conceptual Model with Optimizations

We formalize the conceptual model by creating the structural model for the algorithmic computations. During the formalization, we introduce a set of **optional** optimizations to simplify the structural model, in order to:

1. Make the formal model conform to the *Composition Pattern* to improve the human readability, without interfering the run time behaviour;
2. Simplify the computer readable array-of-integer (AOI) representation to reduce resource consumption and to improve run time efficiency.

It is worth noting that any optimization applied in the formalization process **must not** compromise the computation behaviour. The design should always meet the functional requirements.

Optimization 1: Removing composite blocks containing single child block

In Section 2.4, we suggested to remove the containment if an F-block contains only one child F-block for simplification. For the same reason, we remove the containment brought in by the `communication mediation` capability in the formal model. As a consequence, an F-block and its port maps can be eliminated.

In the CPS software framework, we provided a `metadata` field[4] to identify F-blocks and D-blocks, therefore a simplified block can still be identified as a task, a capability or a function after the containment removal.

Optimization 2: Recomposing function blocks that are triggered in fixed order or in the same composition level

Creating composite blocks for computation could help to minimize reconfiguration and rescheduling effort. For instance, the `orientation detection` capability and the `LED pattern visualization` capability contribute to the scheduled behavioural computation. In practice, these two capabilities can be merged in one composite F-block.

[4]Please refer to Listing 4.1 and Listing 4.2 for the C struct and the examples.

Listing 6.4: The optimized event loop for the microcontroller-based gadgets in standalone mode.

```
When the system is triggered
do event_loop{
    communication_mediator()
    coordinator()
    configurator()
    scheduling::led_control(){
      computation::data_acquisition()
      computation::matrix_multiplication()
      computation::roll_pitch_calculation()
      computation::pattern_generation()
      computation::LED_driver()
    }
}
```

The above optimization practically reduces the amount of memory consumed in the design, but the computation behaviour remains the same. Listing 6.4 shows the *optimized event loop* based on that shown in Listing 6.1.

6.4.1 Formal Structural Model

With the above optimizations, we create the structural model as depicted in Fig. 6.5. The rooted tree graph of the F-blocks is demonstrated by Fig. 6.6. The functions to be assigned in F-blocks are listed in Table 6.1, and the data assigned to D-blocks are listed in Table 6.2.

As the gadget has one simple task, we skip the coordinator between tasks layer (L3) and capabilities layer (L2) and implement one coordinator using the LCSM for simplicity.

6.4.2 F-Blocks Types and Functions

There are eight **computation function blocks** (CFBs): `FB(2)`, `FB(3)`, `FB(4)`, `FB(6)`, `FB(7)`, `FB(8)`, `FB(9)` and `FB(10)`; and two **composite computation blocks** (CPBs): `FB(1)` and `FB(5)`.

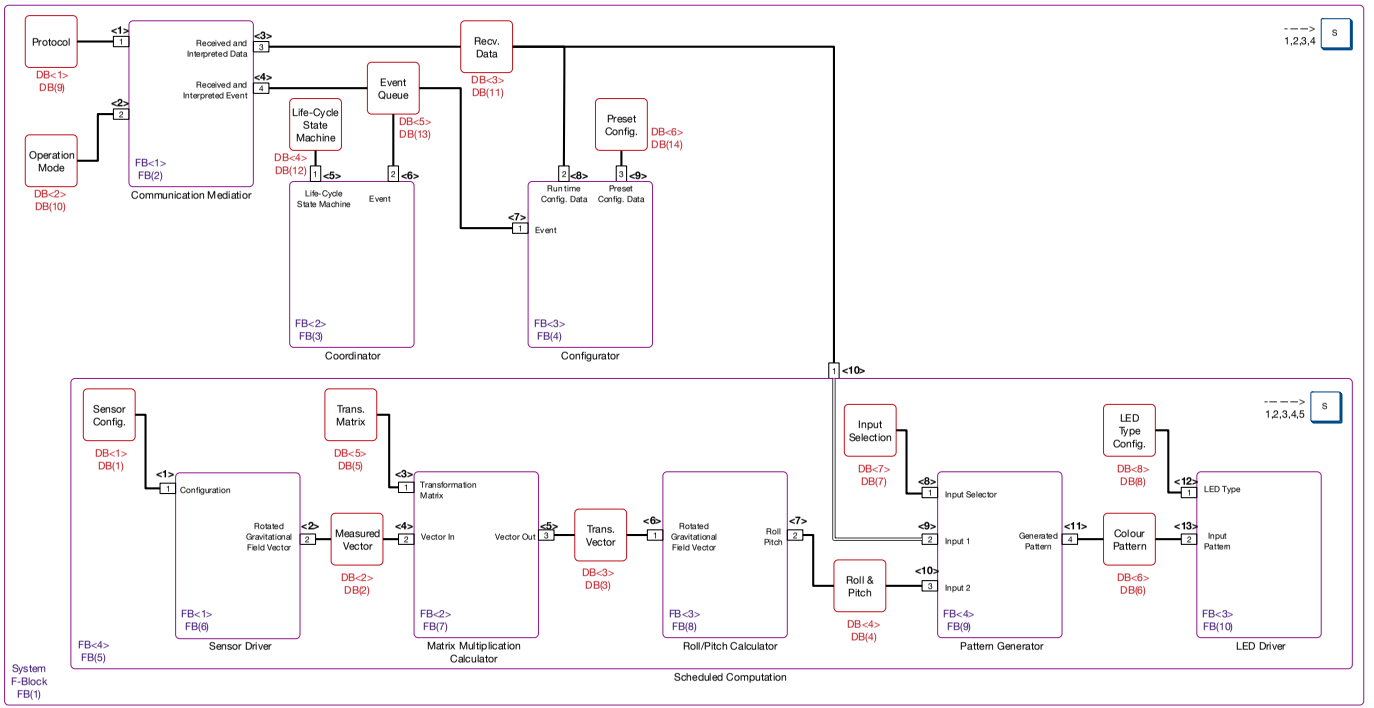

Figure 6.5: The structural model of the gadget system. The functions to be assigned in F-blocks are listed in Table 6.1, and the data to be assigned in D-blocks are listed in Table 6.2.

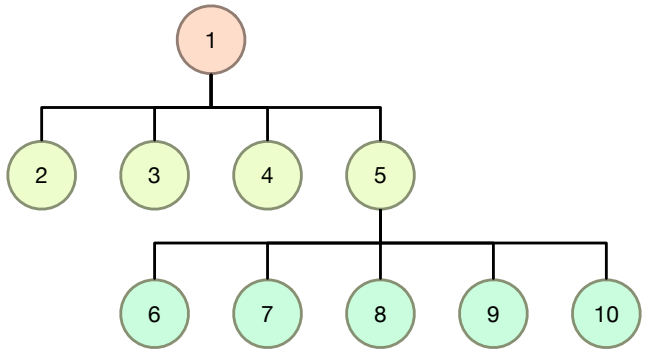

Figure 6.6: The rooted tree graph for F-blocks in the structural model in Fig. 6.5.

Table 6.1: F-blocks types and functions in the structural model

F-block	Type	Function or composition
FB(1)	Composite	System root block
FB(2)	Function	Communication mediator
FB(3)	Function	Coordinator
FB(4)	Function	Configurator
FB(5)	Composite	Scheduled computations
FB(6)	Function	Sensor driver
FB(7)	Function	Matrix multiplication calculator
FB(8)	Function	Roll/pitch calculator
FB(9)	Function	Pattern generator
FB(10)	Function	LED driver

Computation Function Blocks

`FB(2)` is assigned with the `communication mediator` that monitors and handles the incoming and outgoing messages as well as data logging.

`FB(3)` contains the coordinator that operates the Life-Cycle State Machine. It receives external events from the `communication mediator` to coordinate the computation.

`FB(4)` is assigned with a configurator that (re)configures the computation in composite F-block `FB(5)`. The configuration is carried out only when the current state of the Life-Cycle State Machine is `capability configuration`.

`FB(6)` drives the accelerometer to acquire the rotated gravitational field vector.

`FB(7)` calculates the transformed gravitational field vector with respect to the transformation matrix (determined by how the accelerometer is physically mounted on the gadget).

`FB(8)` calculates the orientation in the form of roll and pitch.

`FB(9)` generates colour patterns with respect to the orientation data from one of the two inputs.

`FB(10)` drives the LED ring to show the colour pattern, with respect to the LED device type.

Composite Computation Blocks

`FB(1)` contains four child F-blocks. These child F-blocks are triggered with respect to the event loop in Listing 6.4: `FB(2) communication mediator`, `FB(3) coordinator`, `FB(4) configurator` and `FB(5) scheduled computation`. The order is determined by the scheduling list of `FB(1)`, which is in this case, should be set to {1, 2, 3, 4} when the system, or to be precious, the LCSM is in `Running` state. If the LCSM is in `pausing` or other state, the scheduling list of `FB(1)` is set to {1, 2, 3} so that the `scheduled computation` is excluded from the execution.

`FB(5)` contains all the `computations` for the desired behaviour to carry out the `Local&Remote LED Control` task. The scheduling list of `FB(5)` is determined by the desired capability and the current state of the LCSM. In standalone mode, if the current state of the LCSM is `running`, `FB(5)` will be triggered in `FB(1)`, and the scheduling list of `FB(5)` is set to {1, 2, 3, 4, 5}, which means, the ring LED colour pattern is determined by the orientation of the local IMU integrated in the gadget. In peer-to-peer mode, the scheduling list of `FB(5)` on the `receiver` gadget is set to {4, 5} to skip the local orientation detection.

6.4.3 Data in D-blocks

Data to be stored in D-blocks are listed in Table 6.2.

`DB(1)` stores the I^2C address and the status of the sensor: either initialized or uninitialized. The initialization is carried out when the LCSM is in `resource configuration` state.

Table 6.2: Data stored in the D-blocks in the structural model

D-block	Data
DB(1)	Sensor configuration
DB(2)	Measured vector
DB(3)	Transformed vector
DB(4)	Calculated roll and pitch
DB(5)	Transformation matrix
DB(6)	Desired pattern for LEDs
DB(7)	Input selection
DB(8)	LED type configuration
DB(9)	Communication protocol
DB(10)	Operation mode
DB(11)	Received data
DB(12)	Life-Cycle State Machine
DB(13)	System event queue
DB(14)	Preset configurations

`DB(2)` holds the measured *rotated gravitational field vector* updated by the `sensor driver` in `FB(6)`.

`DB(3)` contains the transformed*rotated gravitational field vector*.

`DB(4)` stores the calculated roll and pitch of a gadget.

`DB(5)` keeps the transformation matrix needed as system-specific knowledge bridging the orientation between the accelerometer and the gadget.

`DB(6)` possesses the generated pattern for the `LED driver` in `FB(10)`.

`DB(7)` indicates which of the two inputs is used on `FB(9)` for pattern generation.

`DB(8)` is the place where the LED type is stored, i.e. the number of LEDs on the LED device.

`DB(9)` holds the communication protocol.

`DB(10)` keeps the current operation mode of the gadget.

`DB(11)` works as a data buffer. The `communication mediator` may receive either run time reconfiguration data for the `coordinator` in `FB(4)`, or the orientation of another gadget for the `pattern generator` in `FB(9)`.

`DB(12)` is reserved for the Life-Cycle State Machine[5] for the `coordinator` in `FB(3)`.

`DB(13)` shares the event queue with the `communication mediator`, the `coordinator` and the `configurator`. The events in the event queue are *consumed* once they are handled by the coordinator `FB(3)`.

`DB(14)` stores the preset configuration data and scheduling lists for self-reconfiguration and run time rescheduling.

6.5 Computer Readable AOI Representation

The computer readable AOI representation of the structural model including the containment and connectivity arrays are listed below:

Containment

CT={ 0, 1, 1, 1, 1, 5, 5, 5, 5, 5 }

Connectivity

CN(1)={ [1, 1], [2, 2], [3, 3], [8, 3], [10, 3], [5, 4], [4, 5], [6, 5], [7, 5], [9, 6] }

CN(5)={ [1, 1], [2, 2], [4, 2], [5, 3], [6, 3], [7, 4], [10, 4], [3, 5], [11, 6], [13, 6], [8, 7], [12, 8] }

PM(5)={ [9, 1] }

6.6 Implementation

The containment and connectivity arrays in the AOI representation are recipes to create the structure of the computation behaviour in the CPS software framework. The functions are implemented in accordance with the discussion in Section 6.4.2, using the CPS function template in Listing 4.9 discussed in Chapter 4.

[5]Please refer to Section 3.3 for the LCSM model and its operation.

It is worth noting that a *light-weight Simple Serial Interface (SSI)* protocol [82] was implemented in C language for efficient binary[6] inter-system communication (see Appendix A).

6.7 Deployment

The gadget requires an embedded board with an I^2C interface to access the IMU and the LCD, a digital I/O pin to drive the ring LED, a UART interface for the A7105 transceiver module to set up wireless communication, and it would be the best to have an on-board debug interface. In this application, we chose Arduino Mega 2560 with some additional considerations:

1. **Low cost**

   Arduino Mega 2560 is a popular and low cost device that can be purchased from Amazon or eBay for less than 15 Euro.

2. **Wide range power input**

   The power input on most of the Arduino devices are regulated and protected so that a wide range (6-20V) external DC power can be applied using battery packs. This is a significant advantage comparing with the single-board computers such as Raspberry Pi and BeagleBone Black that need 5V DC power sources.

3. **Limited memory**

   This is actually a challenge as there is no memory management on many microcontroller based devices. Memory leakage or exhaustion may lead to unpredictable behaviour of the system. However, as we have been spoiled by the tolerance of memory issues on platforms with operating systems, it is a good chance to demonstrate how *bad* the code can be written in a software programming class.

6.8 System Operation Workflow

In this section we run through the **operation workflows** of the designed gadgets in both *standalone mode* and *peer-to-peer mode*. These workflows are

[6] We also recommend to use standardized binary representation in communication such as CBOR (RFC 7049 Concise Binary Object Representation) [18], and CoAP (the RFC 7252 Constrained Application Protocol) [19].

generic and typical when bringing up embedded system-of-systems using the CPS software framework powered by the Life-Cycle State Machine.

6.8.1 Single System Run Time

At power-on, the following operations are carried out on a gadget:

1. The structure of the system is created using the containment and connectivity arrays.
2. The Life-Cycle State Machine is created and stored in `DB(5)`.
3. The Life-Cycle State Machine is initialized, and the current state is `resource creation`.
4. The communication, coordination and configuration functions are deployed on F-blocks `FB(2)`, `FB(3)` and `FB(4)` respectively.
5. `FB(2)` logs the orientation in `DB(4)` and display the data on the LCD.
6. The computation functions are deployed on F-blocks `FB(6)`, `FB(7)`, `FB(8)`, `FB(9)` and `FB(10)` respectively.
7. The scheduling list of `FB(1)` is set to {1, 2, 3}, so that the triggering order is `communication mediator`, `coordinator` and `configurator`. The computation is skipped because the current state of the Life-Cycle State Machine is not `running` yet.
8. `FB(1)` is triggered in an infinite loop, consequently triggering `FB(2)`, `FB(3)` and `FB(4)`.
9. The `coordinator` generates an internal event to trigger the state machine, shifting the current state to `resource configuration`.
10. `DB(1)` is initialized by the `configurator` with an array `[0x00, 0x53]`: the first element indicates that the accelerometer is not configured yet; and the second element is the device address of the accelerometer.
11. `DB(9)` is configured by the `configurator` with the communication protocol and `DB(10)` is set to `0x00` representing "standalone" operation mode.
12. The `coordinator` generates an internal event to trigger the state machine, the current state moves to `capability configuration`.

13. The scheduling list of `FB(5)` is set to {1, 2, 3} by the `configurator` for the standalone operating mode.

14. `DB(3)` is initialized by the `configurator` with an integer 0, indicating that the local orientation will be used to calculate the pattern in standalone mode.

15. The `coordinator` generates an *event queue* to trigger the state machine, shifting the current state to `pausing` and then to `running`.

16. The `configurator` reconfigures the scheduling list of `FB(1)` and set it to {1, 2, 3, 4} to include the triggering of `FB(5)`.

The gadget is now running in standalone mode as a single system.

6.8.2 System-of-Systems Interaction

As a single system, the gadget software looks more complicated than those with ad-hoc implementation. However, the additional complexity, which is, brought in by using the software framework, by decoupling behavioural concerns from structure and by separating configuration and computation, would help integrating single systems into system-of-systems for the following reason: coordination and configuration of behavioural computations are explicitly decoupled so that computations can be updated independently; and configuration can be carried out on different levels in a system. Consequently, the system model becomes *composable*, for single system and system-of-systems.

The separation of coordination, configuration and computation is the key for composing single systems as system-of-systems, and one of the possible inter-system interaction channel is the `communication mediator`. In this section we briefly discuss the system-of-systems interaction as depicted in Fig. 6.1 at the beginning of this chapter.

The central computer is the coordinator in the system-of-systems. It initiates the reconfiguration and rescheduling of the computations on the gadgets to realize the inter-system communication. The computer is interfaced with an A7105 transceiver to join the wireless communication network, it runs a graphical user interface (GUI) that allows users to send or receive messages in the form of `packets` using the SSI protocol (see Appendix A).

Denoting the two gadgets as $\mathbb{G}(\mathbb{1})$ and $\mathbb{G}(\mathbb{2})$, we suppose that both $\mathbb{G}(\mathbb{1})$ and $\mathbb{G}(\mathbb{2})$ are already running in standalone mode at power-on, after *operation 16* in the previous section. To realize the peer-to-peer communication between

$\mathbb{G}(\mathbb{1})$ and $\mathbb{G}(\mathbb{2})$, we need to reconfigure the gadgets by sending corresponding messages from the computer to both of them. We will not discuss the packet details but provide a possible procedure as below.

- Reconfigure $\mathbb{G}(\mathbb{1})$ to peer-to-peer mode, as a `receiver`
  1. Switch the current state of the Life-Cycle State Machine to `capability configuration`;
  2. Reconfigure `FB(9) pattern generator` by `DB(7) input selection`, to use `DB(11) received data` for the pattern generation;
  3. Skip the triggering of `FB(6)`, `FB(7)` and `FB(8)` by rescheduling the scheduling list in `FB(5)`;
  4. Reconfigure the operation mode, update `DB(10)` by `0x10` for `receiver` in peer-to-peer mode;
  5. Switch the current state of the Life-Cycle State Machine back to `running` and the `communication mediator` will send the query request to $\mathbb{G}(\mathbb{2})$.
- Reconfigure $\mathbb{G}(\mathbb{2})$ to send the orientation as a `sender`
  1. Switch the current state of the Life-Cycle State Machine to `capability configuration`;
  2. Reconfigure the operation mode, update `DB(10)` by `0x11` for `sender` in peer-to-peer mode;
  3. Switch the current state of the Life-Cycle State Machine back to `running`.
  4. `FB(2) communication mediator` sends out the measured orientation to $\mathbb{G}(\mathbb{1})$ when receiving the query request.

Now $\mathbb{G}(\mathbb{1})$ and $\mathbb{G}(\mathbb{2})$ are both running in peer-to-peer mode.

If the communication is terminated unexpectedly, after 20 seconds a time-out event will be raised by the `communication mediator` and processed by the `coordinator`, both $\mathbb{G}(\mathbb{1})$ and $\mathbb{G}(\mathbb{2})$ will return to the standalone mode:

1. Switch the current state of the Life-Cycle State Machine to `capability configuration`;
2. Reconfigure the operation mode, update `DB(10)` by `0x00` for standalone mode;

3. Revert the triggering of `FB(6)`, `FB(7)` and `FB(8)` by rescheduling the scheduling list in `FB(5)`;
4. Switch the current state of the Life-Cycle State Machine back to `running`.

Besides the peer-to-peer inter-system communication, $\mathbb{G}(\mathbb{1})$ and $\mathbb{G}(\mathbb{2})$ may run local functions independently. For instance, $\mathbb{G}(\mathbb{1})$ can be configured to log the received orientation on its LCD; and $\mathbb{G}(\mathbb{2})$ could be logging the current state of the Life-Cycle State Machine instead of the gadget orientation.

It is worth noting that $\mathbb{G}(\mathbb{1})$ and $\mathbb{G}(\mathbb{2})$ are running exactly the same code with the same computation behaviour. However, they can be configured to work independently, or to participate in inter-system computation and meanwhile carry out something specific as individual systems.

6.9 Educational Robot Setups

In this section we briefly introduce a selection of the educational robot setups built during this thesis research, demonstrated by Fig. 6.7.

Arduino Car

The Arduino car in Fig. 6.7 (a) was used in the lectures of *Embedded Control System*. We provided a set of system design template code in the CPS software framework for the Arduino car to the students with few programming skills, and eventually got the following positive feedback (referred from the mailing list of the Embedded Control System lecture `ecs@ls.kuleuven.be`):

This project was a good intro to the programming structure with function blocks and data blocks that is used in control software. We now also have a basic knowledge of the C programming language and the use of an Arduino controller since we did not have any programming background.

2-DOF Pan-Tilt Gadget

The 2-DOF pan-tilt gadget in Fig. 6.7 (b) is a simple, low power consumption FPGA-based cyber-physical system. An IMU is mounted on a simple plate as a joystick to actuate the two joints to tilt the pan in the middle.

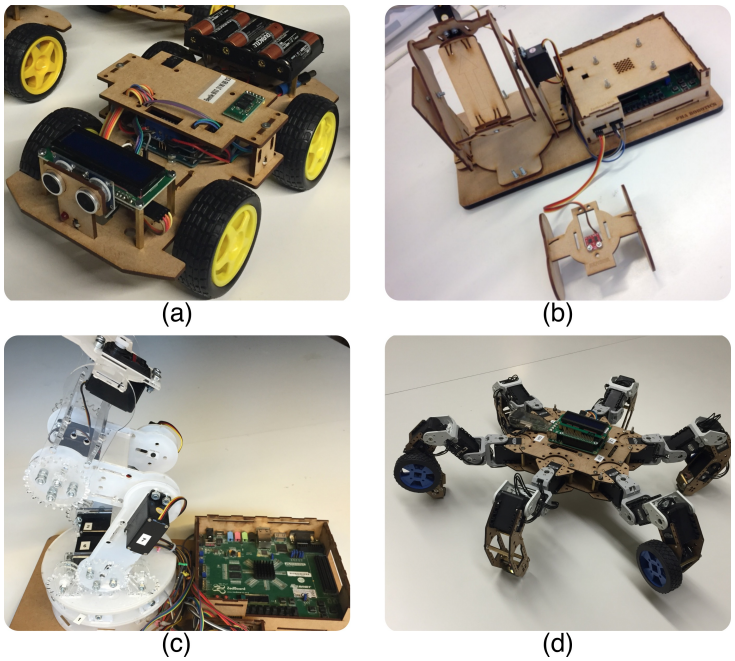

Figure 6.7: Educational setups: (a) Arduino car (b) 2-DOF pan-tilt gadget (c) 5-DOF robot arm (d) Crawler

This setup was used in the *5th European PhD school in Robotic Systems* in Leuven in January 2014, in the introduction sessions of Microblx software framework [69]. In August 2014, the same setup contributed in the seminar of *Embedded Control System on Microcontrollers and FPGA SoC, Sensors and Actuators* organized for the TEMPUS project.

5-DOF Robot Arm

The 5-DOF robot arm in Fig. 6.7 (c) was designed in 2014. The setup was used in a master thesis for IMU data analysis. We hacked the cheap servo motors by soldering an extra piece of wire on the internal potentiometer as an analog position feedback.

Crawler

The crawler in Fig. 6.7 (d) is a 18-DOF wheeled-hexapod that walks with six legs, in which four of them are wheel-equipped. The crawler may walk as a mechanical vehicle like a normal hexapod, and it may also transform to a wheeled vehicle by folding the wheeled legs for fast, low energy consumption moving. It is a complex platform that can be used in student project to explore advanced robotics, kinematics, and programming. The crawler is controlled by a popular Raspberry Pi 3 in the current setup.

6.10 Conclusion

This chapter presented a complete system design process of a microcontroller-based gadget to demonstrate how to use the meta models, design patterns and software discussed and developed in the previous chapters, to model and implement cyber-physical systems and system-of-systems. The gadget is in practice an embedded platform with limited computation resources that helps to test the efficiency and effectiveness of (1) using the CPS software framework to create the structure of computation behaviour, and (2) using the cFSM state machine modelling software to realize pure coordination, by means of the Life-Cycle State Machine, with optional optimizations to improve resource utilization and run time efficiency.

The design process of the gadget covered software programming, hardware selection and digital fabrication. It is a compact teaching material for system design, and a practical example on how to systematically design and build low cost educational setups. We have learned in real classes that students would be well motivated to study system design with simple and concrete examples. The presented setups, the design methodology and the software framework have contributed significantly in the teaching activities, we got quite some positive feedback from the students that inspired us to improve the various teaching materials and educational setups.

The complete implementation of the microcontroller-based gadget, including the software code, a list of used and alternative hardware devices and platforms, as well as the sketch of the mechanical structure for laser cutting are available at https://gitlab.mech.kuleuven.be/u0066910/ces.

Chapter 7

General Conclusions

This chapter concludes the thesis by summarizing and discussing the contributions of the research, as well as its limitations; the latter are accompanied by concrete suggestions about how to extend the presented research to tackle the limitations.

The research objectives, as discussed in the introduction (Section 1.5), are repeated here:

1. **Methodology**: Developing a systematic approach to model cyber-physical systems, and to guide the creation of reusable, flexible and adaptable software for robotic and cyber-physical systems, as well as system-of-systems.
2. **Software framework**: Developing and implementing (a first prototype of) a composable software framework that is consistent with the approach in the first objective.
3. **Software compatibility**: Maximizing the compatibility of the software framework for different embedded device families.
4. **Best practices**: Find the best practices on integrating sensoring, actuating and computing tasks on heterogeneous devices, and on utilizing the available hardware resources, in the context of the systematic approach.
5. **Education**: Using the meta model developed for cyber-physical system design and the software framework, together with the identified best practices, to build robots and setups to demonstrate, introduce and teach system-of-systems design, and to facilitate the educational activities.

7.1 Contributions

This Section summarizes and discusses the contributions of this thesis, and links each of them to one or more of the above-mentioned objectives.

System Design Methodology Using Cyber-Physical Stack Meta Model and Design Patterns

The first contribution of this thesis (in the context of the *Methodology* and *Education* objectives) is the **novel** system design methodology using the developed Cyber-Physical Stack (CPS) **meta model**. The originality of the meta model proposes an orthogonal representation of a system, by describing a cyber-physical system in **four layers** covering tasks, **capabilities**, functions and devices, from hardware to software. The formal separation of the four layers represents the **consolidation** of the **design patterns**: the Composition Pattern and the Coordination-Configuration Pattern in the Cyber-Physical Stack. The research has motivated by system-level arguments, and illustrated by concrete examples, that both patterns are equally applicable and effective to model and design capabilities and functions in system design, as they were in their original context of task-level models.

The four-layer model structure (also called the "Cyber-Physical Stack") is an instance of a *layered architecture* as discussed in [107], with four concrete *levels of abstraction* that are simple enough to motivate and understand their role in the complete system, yet rich enough to provide all possible types of system and system-of-systems architectures, and, in addition, designed to explicitly avoid the *information hiding* between layers that limits the reactivity in most architectures. In addition to the widely used *tasks* and *functions* software layers [133, 11], plus the classical *devices* hardware layer [32], we explicitly defined and introduced the *capabilities* layer as the novel "first-class citizen" in the CPS architecture, to answer the questions about "how-to create" and "how-to use" capabilities, which were not covered in the literature, [17] and [41]. The former was solved by introducing *system-dependent knowledge* in the form of configuration parameters to (existing) *system-independent* functions; and the later was tackled by the *event loop meta model*.

Event Loop Pattern for Single- and Multi-Threaded Deployments

The event loop meta model is a concrete way to realise the structural (de)composition of *computational behaviours*. It supports, both, the **scalability of structure** in a system (e.g., by means of the AOI representation for containment of sub-systems, their connectivity and their scheduling), and the **scalability of behaviour**, from single- to multi-threaded deployments, and from single process to multi-computer deployments.

Composable Finite State Machine Meta Model and Life Cycle State Machine Coordinating Model

This contribution (fitting in the objectives of *Best Practices*, *Methodology* and *Education*) models the **coordination of computation behaviours** of cyber-physical systems, by means of the *composable Finite State Machine* (cFSM) meta model. We implemented the 4^{th} generation of the Life-Cycle State Machine (LCSM) using cFSM on the basis of the work presented by Soetens [119], Klotzbücher et al. [68] and Vanthienen et al. [127]; the major improvement come from revealing the **resource-capability** relationship in the CPS meta model, i.e., the behaviour of capabilities on a higher layer depend on, and are continuously influenced by, coordinating, configuring and scheduling the resources on a lower layer.

The LCSM is an essential coordinating tool since it explicitly decouples computation in the capabilities, from coordination of the required resources and the coordination of the executed tasks. Consequently, it improves the reusability of both task and resource models and components. In the system-of-systems context, the LCSM also improves the composability of individual sub-systems on a higher level of composition, because sub-systems are in fact just "resources" contributing to the "capabilities" of the complex system.

A Full System-of-Systems Design Example

We discussed a complete system-of-systems design process of a microcontroller-based gadget as a concrete example in Chapter 6, in which both the CPS and cFSM meta models as well as the design patterns are involved. This example contributes to the objectives of *Software Framework* and *Education*.

We summarized two *workflows* in the example: (1) an *operation workflow* of single system and system-of-systems at run time; and (2) a brief *design workflow* of the system design methodology.

The *operation workflow* of the gadgets practically brought in a convincing reason of using the CPS meta model and software framework in complex system design, instead of ad-hoc implementation: **decoupling behavioural concerns and separating configuration and computation would help integrating single systems into system-of-systems**. To facilitate the integration on system-of-systems level for **recomposability**, **run time rescheduling** and **autonomous self-reconfiguration**, we need to introduce necessary additional complexity and spend extra resources and implementation efforts on sub-system level using the software framework.

The *design workflow* is not only a roadmap to build a cyber-physical system using the design methodology, but also a set of *waypoints* that allow or even suggest designers to review the modelling progress and to apply necessary optimizations. This set of waypoints are concrete and reusable in the sense that they are *anchored* in the Cyber-Physical Stack meta model.

The educational values integrated in the microcontroller-based gadget together with the above discussed workflows are significant: the concrete example could be extended as an introductory material in system design, in which all the hardware are listed and all the software are provided.

The success on motivating students by simple cyber-physical examples such as the Arduino car introduced in Section 6.9 together with the developed methodology and software verified and indicated **reproducibility** and approved the educational and disseminational **impact** of the result of this thesis.

CPS Software Framework and cFSM Software

The CPS software framework and the cFSM software are the two *cores* of the **composable and embeddable software** (CES) contributed by this thesis. The implementation in C language is independent from any platform specific library, implying that the CES is highly *reusable* on different devices.

The CES practically enhanced the **reproducibility** of the CPS and cFSM meta models as a tailored tool. Using the microcontroller-based gadget as a concrete example, one can easily duplicate the system-of-systems since the full recipe, including the software code, the hardware list as well as the mechanical structure sketch, are provided as open hardware and software. The CES and the educational robot setups are already serving in the Embedded Control

System class at KU Leuven, contributing to the teaching activities and software programming practical sessions.

Best Practices of Using Embedded Device Resources in System Design

This contribution involves the optimizations in system design to reduce resource consumption and human effort while retaining and improving computation performance and efficiency at run time. The best practices cover classical sequential processor based embedded platforms and FPGA System-on-Chip (SoC) with concurrent processing capability.

We used *embeddability* as an essential benchmark when developing the CPS meta model, the cFSM meta model and the software framework. However, *embedded-friendliness* is not a constraint, but an additional feature that extends the **usability** and **reusability**, so that the meta models and the software are applicable in system design of any computation platforms including devices from the embedded families.

The lessons learned from using FPGA SoC further revealed the importance of **reusability**. We suggested to separate the four major function types on FPGA-based cyber-physical system design, to improve the development efficiency using different programming tools and languages, since a highly reusable computation function can be deployed either on hardware (FPGA fabric) or on software (embedded processors).

7.2 Limitations and Future Work

This section discusses the limitations in the current research, together with concrete solutions to tackle these limitations in future work.

7.2.1 Mature Modelling Tools Are Needed

In this thesis, we developed a systematic methodology to facilitate the design of cyber-physical systems and system-of-systems. The structure of a system's conceptual model can be formalized using the AOI representation, and implemented using the composable and embeddable software (CES). The effectiveness of the above efforts has been proved by the experience obtained from educational activities: most of the students were capable to manage the

system design phases and the AOI representation in a two-hour exercise session, and they were able to start creating the structure of systems using the CES at the end of the same session. However, the developed Cyber-Physical Stack software framework and the composable Finite State Machine software are rather simple: 1) The software focuses on the automation of the **structural aspects** of systems, designers still need to sketch the system manually, which is not convenient when designing large and complex systems; 2) The event loop software architecture for capability realization facilitates designers to compose the behaviour, however, as the event loop is application- and system-specific, although the structure of a system is highly reusable, the reusability of an implemented event loop in software is rather low since it is always coupled with a particular system.

The conceptual and formal modelling could be greatly facilitated by mature tooling in the future with visualization of the structural composition and decomposition, and the event loop meta model could benefit from automated code generation for different systems. The CES is needed as part of the core in the above-mentioned system design tooling, since the implemented software has covered the low-level functions to create the structure. It would be more convenient if the computer readable formal model i.e. the AOI representation for containment and connectivity could be automatically generated using graphical tools.

7.2.2 Consolidation of Meta Modelling Software

The AOI representations in the Cyber-Physical Stack meta model and the composable Finite State Machine meta model are quite similar. The similarity is not a coincidence as both meta models conform to the NPC4 [115]. The implementations of the above meta models are two separate cores in the CES, they can be merged in the future to reduce the amount of code, and more importantly, to strengthen the modelling capability of the software, e.g. adding support in cFSM to model boundary crossing transition using *port map*, which is currently a limitation of the cFSM software.

7.2.3 Simple Scheduler Behaviour

In the Cyber-Physical Stack, the structure of computation behaviour is formulated using the formalized AOI representation. The AOI representation focuses on the relationships between functions and data, i.e. the containment and connectivity of F- and D-blocks. The execution of the F-blocks is scheduled through the scheduling list stored in the S-blocks embedded in the composite

F-blocks. In the current CPS meta model, the behaviour of the scheduling functions assigned to the S-blocks are rather simple: they trigger child F-blocks according to the scheduling lists modelled as 1-D arrays. The triggering mechanism of the F-blocks is *search-in-depth*. On complex systems, designers may have to extend the implementation of the scheduling functions for S-blocks to support *interruptible* scheduling lists. As one of the future works, a multi-functional scheduling function model can be developed as an extension of the current implementation.

7.2.4 "Resource Saving" Policy Brings the Needs of Additional Knowledge

In the formal structure model of the Cyber-Physical Stack, the AOI representation reveals the relationship between *ports*, but it does not explicitly show which F-block does a port belong to. The motivation of doing so is to reduce the amount of numbers needed when representing the structure, so that the AOI representation requires less bytes in practice, and thus fits better on embedded systems such as microcontrollers. Number of ports on each F-block is therefore needed when creating or recreating the system structure. In practice, due to the consideration of embedded-friendliness, extra knowledge is required during the system design process. This is actually a trade-off between "resource saving" and the amount of additional knowledge required. As both resources and knowledge are in the form of *bytes*, it would be an interesting topic as a future work to model this trade-off and find the break-even point.

7.2.5 Communication for System-of-Systems

Communication is an essential capability in any system that may potentially participating in a complex system as a sub-system. In the system design example presented in Chapter 6, the communication function is configured with the SSI protocol on a microcontroller-based gadget in order to exchange messages with "understandable meanings" in the system-of-systems. The protocol and the message format are highly reusable on different systems. Although modelling communication is out of the scope of this thesis, it could be one of the future works to include the communication behaviour in the composable and embeddable software.

Appendix A

A Light-Weight Simple Serial Interface Protocol

We use a light-weight serial communication protocol to facilitate the communication between computation devices. The protocol is a reduced version of the **Simple Serial Interface** (SSI) proposed by MaxixIntegrated [82], which composes messages as data packets in the format shown in Fig. A.1. This protocol assumes that all the devices are interconnected, as a *fully connected network*.

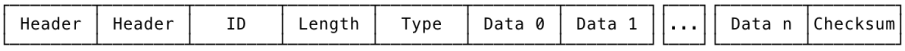

Figure A.1: A packet consists of a two-byte header, a one-byte device ID, a one-byte data length, a one-byte packet type, multiple bytes of data and a one-byte checksum.

Packets in the network are visible to all the devices, a device needs to decide whether the received message should be interpreted or not. It is essential to define packet types by using different header bytes, and it is also crucial to ensure valid packets by appending a checksum byte at the end of a packet.

A.1 Packet Composition

A packet is composed by a 2-byte `header`, a 1-byte `ID`, a 1-byte `packet length`, a 1-byte `opcode` an N-byte `data` and a 1-byte `checksum`.

A.1.1 Header

The first two bytes are the header bytes that can be used to identify valid packets. Users may define particular headers for different types of packets. In this thesis, we define two types of packets: instruction and response.

An *instruction packet* is a request or command actively sent from one device to a target device. The target device may interpret the incoming packet and send a *response packet* to the network.

A.1.2 ID

The third byte specifies the device ID. The device ID could be either a *target device ID* or a *source ID*, with respect to the instruction and response type of a packet.

A.1.3 Length

The fourth byte indicates the packet length. It is determined by the total number of bytes of the data.

A.1.4 Instruction and Response Type

The fifth byte specifies the type of an instruction or a response as an additional information in the communication.

A.1.5 Data

Either an instruction or a response packet may contain a number of data bytes. For instance, an instruction packet from a configurator may carry configuration parameters to change the behaviour of a computation function.

Listing A.1: C struct of the light-weight simple serial interface protocol packet.

```
typedef struct ssi_pkt{
        unsigned short header;
        unsigned char  id;
        unsigned char  length;
        unsigned char  type;
        unsigned char  data[];
} ssi_pkt_t, *ssi_pkt_p
```

A.1.6 Checksum

Checksum helps to detect packet error in the communication. As the length of a packet is indicated by the *packet length* byte, it is easy to locate the checksum since it is appended behind the last data byte by the `sender`.

The checksum is calculated by the `receiver` in the same way to detect transmission error. If the received and calculated checksum bytes differ, the received packet will be considered as invalid.

The checksum is determined by equation A.1:

$$checksum = (ID + Length + Type + \sum_{i=0}^{Length} (Data[i])) \tag{A.1}$$

If the result is greater than hexadecimal `0xFF`, the lowest byte will be used as the checksum byte.

A.2 C Struct of a Packet

The C struct of the light-weight SSI communication packet is defined in Listing A.1:

The two-byte `header` determines the type of the packet; the `id` stores either target or source device ID, with respect to the packet type. The length of the data is stored in `length`.

The data body is declared as a *flexible array* as the last member, so that the size of the data body can be defined freely for different applications and functions. The last byte of `data` is the used by the `checksum` in this case.

A.3 Communication Packets for the Gadgets

This section briefly introduces the communication packets defined for the system-of-systems example in Chapter 6. In the example, an instruction packet may carry a command to change the behaviour of a gadget, for instance switching from standalone mode to peer-to-peer mode; or it may include an event to trigger the Life-Cycle State Machine. A response packet is composed by the `sender` gadget carrying the orientation data and the receiver gadget ID. The header bytes for instruction and response packets, together with the instruction types are listed in Table A.1. The hexadecimal numbers for header bytes and the instruction and response types are arbitrarily determined.

Table A.1: Instruction and response packet headers and types

Packet Type	Header	Header	Instruction and Response Type	Remarks
Instruction	0xCE	0xCE	0x30	LCSM event
			0x31	Gadget operation mode
Response	0xDF	0xDF	0x32	orientation data

There are eight transitions in the Life-Cycle State Machine, each of them is triggered by an event, we define these events using single-byte hexadecimal numbers as shown in Table A.2.

Table A.2: Event-transition table of an LCSM with event IDs

Transition	Transition ID	Event	Event ID
TR_capconf_pau	TR(1)	EV_capconf_pau	0x01
TR_pau_capconf	TR(2)	EV_pau_capconf	0x02
TR_pau_run	TR(3)	EV_pau_run	0x03
TR_run_pau	TR(4)	EV_run_pau	0x04
TR_act_dep	TR(5)	EV_act_dep	0x05
TR_cre_resconf	TR(6)	EV_cre_resconf	0x06
TR_resconf_del	TR(7)	EV_resconf_del	0x07
TR_dep_act	TR(8)	EV_dep_act	0x08

For instance, packet `[0xCE 0xCE 0x01 0x01 0x30 0x04 0xC9]` is an instruction with an *LCSM event* for $\mathbb{G}(\mathbb{1})$ to switch the current state from `Running` to `Pausing`. The first two bytes are the header for instruction packets; the third byte is the target gadget ID `0x01`, i.e. $\mathbb{G}(\mathbb{1})$; the fourth byte shows that the

data length is 1 byte; the fifth byte implies that the packet is a command to operate the LCSM; the sixth byte is the Event ID, in this case, `0x04` for event `EV_run_pau` to trigger transition `TR(4)`. The last byte `0xC9` is the calculated checksum byte for error detection by the `receiver`.

Bibliography

[1] ABB. ABB Robotics. http://www.abb.com/robotics/, 2011. pages 88

[2] Arduino. Arduino Playground: PROGMEM. http://playground.arduino.cc/Main/PROGMEM, 2016. pages 83

[3] Atkinson, C., and Kühne, T. Model-driven development: A metamodeling foundation. *IEEE Software 20*, 5 (Sept. 2003), 36–41. pages 9

[4] Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice*, 3rd ed. Addison-Wesley Professional, 2012. pages 10, 47

[5] Basu, A., Bozga, M., and Sifakis, J. Modeling Heterogeneous Real-time Components in BIP. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods* (Washington, DC, USA, 2006), SEFM '06, IEEE Computer Society, pp. 3–12. pages 11

[6] Bay, H., Ess, A., Tuytelaars, T., and Van Gool, L. Speeded-Up Robust Features (SURF). *Comput. Vis. Image Underst. 110*, 3 (June 2008), 346–359. pages 7

[7] Bento, L. C., Parafita, R., and Nunes, U. Intelligent traffic management at intersections supported by V2V and V2I communications. In *2012 15th International IEEE Conference on Intelligent Transportation Systems* (Sept 2012), pp. 1495–1502. pages 1

[8] Bézivin, J., Paige, R. F., Assmann, U., Rumpe, B., and Schmidt, D. C. Manifesto - model engineering for complex systems. *CoRR abs/1409.6591* (2014). pages 17

[9] Biggs, G., and Macdonald, B. A survey of robot programming systems. In *in Proceedings of the Australasian Conference on Robotics and Automation, CSIRO* (2003), p. 27. pages 11

[10] Borst, P., Akkermans, H., and Top, J. Engineering ontologies. *Int. J. Hum.-Comput. Stud. 46*, 2-3 (Mar. 1997), 365–406. pages 17

[11] Brehmer, M., and Munzner, T. A multi-level typology of abstract visualization tasks. *IEEE Transactions on Visualization and Computer Graphics 19*, 12 (Dec. 2013), 2376–2385. pages 7, 132

[12] Broenink, J. F. Introduction to physical systems modeling with bond graphs. In *in the SiE whitebook on Simulation Methodologies* (1999), p. 2. pages 18

[13] Bruyninckx, H. Open RObot COntrol Software. http://www.orocos.org/, 2001. Last visited June 2011. pages 14, 89

[14] Bruyninckx, H. Open robot control software: the OROCOS project. In *2001 IEEE International Conference on Robotics and Automation (ICRA)* (2001), vol. 3, pp. 2523–2528. pages 7

[15] Bruyninckx, H. Embedded control systems (h04p5). http://people.mech.kuleuven.be/~bruyninc/ecs, 2016. pages 14

[16] Bruyninckx, H., Klotzbücher, M., Hochgeschwender, N., Kraetzschmar, G., Gherardi, L., and Brugali, D. The BRICS component model: A model-based development paradigm for complex robotics software systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing* (New York, NY, USA, 2013), SAC '13, ACM, pp. 1758–1764. pages 22

[17] Buehler, J. Capabilities in heterogeneous multi robot systems. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence* (2013), IJCAI '13, AAAI Press, pp. 3207–3208. pages 7, 24, 132

[18] Carsten, B. CBOR, RFC 7049 Concise Binary Object Representation, 2014. pages 123

[19] Carsten, B. CoAP, RFC 7252 Constrained Application Protocol, 2014. pages 123

[20] Coffman, E. G. *Formalism in Computer System Design: Models of Parallelism and Concurrency.* Department Technical Report Series. Computing Laboratory, University of Newcastle upon Tyne, 1969. pages 10

[21] Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., and Zhang, Z. High-Level Synthesis for FPGAs: From Prototyping to

Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 30*, 4 (April 2011), 473–491. pages 96

[22] Crick, C., Jay, G., Osentoski, S., and Jenkins, O. C. ROS and Rosbridge: Roboticists out of the loop. In *Proceedings of the Seventh Annual ACM/IEEE International Conference on Human-Robot Interaction* (New York, NY, USA, 2012), HRI '12, ACM, pp. 493–494. pages 7

[23] Crick, C., Jay, G., Osentoski, S., Pitzer, B., and Jenkins, O. C. *Rosbridge: ROS for Non-ROS Users*. Springer International Publishing, Cham, 2017, pp. 493–504. pages 7

[24] De Koster, J., Marr, S., Van Cutsem, T., and D'Hondt, T. Domains: Sharing state in the communicating event-loop actor model. *Computer Languages, Systems & Structures 45* (2016), 132–160. pages 47

[25] De Niz, D. Diagrams and languages for model-based software engineering of embedded systems: UML and AADL. *White Paper, http://www.sei.cmu.edu/library* (2007). pages 8

[26] De Schutter, J., De Laet, T., Rutgeerts, J., Decre, W., Smits, R., Aertbelië, E., Claes, K., and Bruyninckx, H. Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty. *The International Journal of Robotics Research 26 5* (2007), 433–455. pages 57

[27] Deconinck, G., Labeeuw, W., Vandael, S., Beitollahi, H., Craemer, K. D., Duan, R., Qui, Z., Ramaswamy, P. C., Meerssche, B. V., Vervenne, I., and Belmans, R. Communication overlays and agents for dependable smart power grids. In *Critical Infrastructure (CRIS), 2010 5th International Conference on* (Sept 2010), pp. 1–7. pages 2

[28] Decré, W. *Optimization-Based Robot Programming with Application to Human-Robot Interaction*. PhD thesis, Departement Werktuigkunde, Faculty of Engineering, KU Leuven, July 2011. De Schutter, Joris and Bruyninckx, Herman (supervisors). pages 2

[29] Denolf, K., Neuendorffer, S., and Vissers, K. Using C-to-gates to program streaming image processing kernels efficiently on FPGAs. In *2009 International Conference on Field Programmable Logic and Applications* (Aug 2009), pp. 626–630. pages 97

[30] Desmouliers, C., Oruklu, E., Aslan, S., Saniie, J., and Vallina, F. M. Image and video processing platform for field programmable gate arrays using a high-level synthesis. *IET Computers Digital Techniques 6*, 6 (November 2012), 414–425. pages 97

[31] Dickerson, S. L., and Lapin, B. D. Control of an omni-directional robotic vehicle with mecanum wheels. In *Telesystems Conference, 1991. Proceedings. Vol.1., NTC '91., National* (Mar 1991), pp. 323–328. pages 97

[32] Dijkstra, E. W. Classic operating systems. In *Communications of the ACM*, P. Brinch Hansen, Ed. Springer-Verlag New York, Inc., New York, NY, USA, 1968, ch. The Structure of the THE Multiprogramming System, pp. 223–236. pages 6, 132

[33] Doroodgar, B., Ficocelli, M., Mobedi, B., and Nejat, G. The search for survivors: Cooperative human-robot interaction in search and rescue environments using semi-autonomous robots. In *2010 IEEE International Conference on Robotics and Automation (ICRA)* (May 2010), pp. 2858–2863. pages 5

[34] Douglass, B. P. *Design Patterns for Embedded Systems in C: An Embedded Software Engineering Toolkit*, 1st ed. Newnes, Newton, MA, USA, 2010. pages 16

[35] Eker, J., Janneck, J. W., Lee, E. A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., and Xiong, Y. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE 91*, 1 (Jan 2003), 127–144. pages 11

[36] El-Zaher, M., Dafflon, B., and Gechter, F. A cyber-physical model for platoon system. In *2015 9th International Conference on Software, Knowledge, Information Management and Applications (SKIMA)* (Dec 2015), pp. 1–8. pages 2

[37] Er, M. J., and Deng, C. Obstacle avoidance of a mobile robot using hybrid learning approach. *IEEE Transactions on Industrial Electronics 52*, 3 (June 2005), 898–905. pages 4

[38] Erickson, J. Part IV - Graphs, Lecture 18: Basic Graph Algorithms. In *Algorithms and Models of Computation*. http://jeffe.cs.illinois.edu/teaching/algorithms/, 2015, pp. 398 – 411. pages 64

[39] Fernández, M. *Models of Computation - An Introduction to Computability Theory.* Undergraduate Topics in Computer Science. Springer, 2009. pages 23

[40] Fritzson, P. *Introduction to Modeling and Simulation.* John Wiley & Sons, Inc., 2014, pp. 1–18. pages 13

[41] Fua, C. H., and Ge, S. S. COBOS: Cooperative backoff adaptive scheme for multirobot task allocation. *IEEE Transactions on Robotics 21*, 6 (Dec 2005), 1168–1178. pages 24, 132

[42] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. pages 16

[43] Garcia, F., Diez, E., and L., F. Design of a high precision digital tachometer for speed and position measurement. In *Design of Integrated Circuits and Systems DCIS 2000* (Montpellier, France, 2000), vol. 15, pp. 1–9. pages 94

[44] Gerkey, B., Vaughan, R., Howard, A., and Koenig, N. The Player/Stage Project. http://playerstage.sourceforge.net/, 2001. pages 89

[45] Gershenfeld, N. *Fab: The Coming Revolution on Your Desktop – from Personal Computers to Personal Fabrication*. Basic Books, Inc., New York, NY, USA, 2007. pages 6

[46] Gruber, T. R. A translation approach to portable ontology specifications. *Knowledge Acquisition 5*, 2 (June 1993), 199–220. pages 17

[47] Gu, J. S., and de Silva, C. W. Development and implementation of a real-time open-architecture control system for industrial robot systems. *Engineering Applications of Artificial Intelligence 17*, 5 (2004), 469–483. pages 89

[48] Guo, Y., McCain, D., Cavallaro, J. R., and Takach, A. Rapid Industrial Prototyping and SoC Design of 3G/4G Wireless Systems Using an HLS Methodology. *EURASIP Journal on Embedded Systems 2006*, 1 (2006), 1–25. pages 97

[49] Harary, F., and Palmer, E. M. Chapter 3 - Trees. In *Graphical Enumeration*, F. Harary and E. M. Palmer, Eds. Academic Press, 1973, pp. 51 – 80. pages 32, 60

[50] Harel, D. Statecharts: A Visual Formalism For Complex Systems. *Science of Computer Programming* (1987), 231–274. pages 10, 58

[51] Hebert, B., Brule, M., and Dessaint, L.-A. A high efficiency interface for a biphase incremental encoder with error detection. *ITIE 40*, 1 (1993), 155–156. pages 93

[52] Henderson, P., and Walters, R. System design validation using formal models. In *Rapid System Prototyping, 1999. IEEE International Workshop on* (Jul 1999), pp. 10–14. pages 10

[53] Henning, M. A. A survey of selected recent results on total domination in graphs. *Discrete Mathematics 309*, 1 (2009), 32 – 63. pages 32

[54] Henzinger, T. A. *The Theory of Hybrid Automata*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 265–292. pages 10

[55] Hrúz, B., and Zhou, M. *Modeling and Control of Discrete-event Dynamic Systems: with Petri Nets and Other Tools*. Advanced Textbooks in Control and Signal Processing. Springer London, 2007. pages 10

[56] Inc., X. Vivado Design Suite - HLx Editions. https://www.xilinx.com/products/design-tools/vivado.html, 2016. pages 10

[57] Instruments, N. Getting Started with LINX. https://www.labviewmakerhub.com, 2015. pages 12

[58] Iwanaga, N., Abe, T., and Yamawaki, A. Development of fixed-point trigonometric function library for high-level synthesis. In *International Conference on Intelligent Systems and Image Processing* (2013). pages 100

[59] Kahng, A. B., Lienig, J., Markov, I. L., and Hu, J. *Timing Closure*. Springer Netherlands, Dordrecht, 2011, pp. 219–264. pages 87

[60] Karcanias, N., and Hessami, A. G. System of systems and emergence part 1: Principles and framework. In *2011 Fourth International Conference on Emerging Trends in Engineering Technology* (Nov 2011), pp. 27–32. pages 1

[61] Kehoe, B., Kahn, G., Mahler, J., Kim, J., Lee, A., Lee, A., Nakagawa, K., Patil, S., Boyd, W. D., Abbeel, P., and Goldberg, K. Autonomous multilateral debridement with the raven surgical robot. In *2014 IEEE International Conference on Robotics and Automation (ICRA)* (May 2014), pp. 1432–1439. pages 5

[62] Keller, R. M. Formal verification of parallel programs. *Commun. ACM 19*, 7 (July 1976), 371–384. pages 10

[63] Kindratenko, V. V., and Brunner, R. J. Accelerating Cosmological Data Analysis with FPGAs. In *17th IEEE Symposium on Field Programmable Custom Computing Machines, 2009.* (April 2009), pp. 11–18. pages 97

[64] Klinker, G., Bhola, C., Dallemagne, G., Marques, D., and McDermott, J. Usable and reusable programming constructs. *Knowledge Acquisition 3*, 2 (1991), 117 – 135. pages 25

[65] Klotzbücher, M. *Domain Specific Languages for Hard Real-Time Safe Coordination of Robot and Machine Tool Systems.* PhD thesis, Departement Werktuigkunde, Faculty of Engineering, KU Leuven, April 2013. Bruyninckx, Herman (supervisor). pages 9, 57

[66] Klotzbücher, M., Biggs, G., and Bruyninckx, H. Pure coordination using the coordinator–configurator pattern. *CoRR abs/1303.0066* (2013). pages 11, 23, 57, 66

[67] Klotzbücher, M., and Bruyninckx, H. Hard Real-Time Control and Coordination of Robot Tasks using Lua. In *Proceedings of the Thirteenth Real-Time Linux Workshop* (July 2011), pp. 37–43. pages 22

[68] Klotzbücher, M., and Bruyninckx, H. Coordinating Robotic Tasks and Systems with rFSM Statecharts. *JOSER 3* (2012), 28–56. pages 19, 57, 58, 59, 133

[69] Klotzbücher, M., and Bruyninckx, H. Microblx: a Reflective, Real-Time Safe, Embedded Function Block Framework. In *Proceedings of the Fifteenth Real-Time Linux Workshop* (Switzerland, 2013). pages 128

[70] Kordon, F., Hugues, J., Canals, A., and Dohet, A. *Embedded Systems: Analysis and Modeling with SysML, UML and AADL*, 1st ed. Wiley-IEEE Press, 2013. pages 11

[71] Kortenkamp, D., and Simmons, R. *Robotic Systems Architectures and Programming.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 187–206. pages 10

[72] Kube, C. R. Task modelling in collective robotics. *Autonomous Robots 4*, 1 (1997), 53–72. pages 26

[73] KUKA. KUKA N.V. http://www.kuka.be/, 2007. pages 88

[74] Lane, J. A. What is a System of Systems and Why Should I Care? *Technical Report, University of Southern California* (2013). pages 1

[75] Lee, E. A. Cyber-physical systems: A rehash or a new intellectual challenge?, June 2013. Invited Talk in the Distinguished Speaker Series, sponsored by the IEEE Council on Electronic Design Automation (CEDA) held at the Design Automation Conference (DAC), Austin, Texas. pages 2

[76] Li, D., and Cui, X. The research on the comparison of several embedded microprocessors and its development trend. In *Chinese Automation Congress (CAC), 2013* (Nov 2013), pp. 818–823. pages 4

[77] Li, M., Batmaz, F., Guan, L., Grigg, A., Ingham, M., and Bull, P. Model-based systems engineering with requirements variability for embedded real-time systems. In *Model-Driven Requirements Engineering Workshop (MoDRE), 2015 IEEE International* (Aug 2015), pp. 1–10. pages 11

[78] Li, X., Fan, Y., Madnick, S., and Sheng, Q. Z. A pattern-based approach to protocol mediation for web services composition. *Information and Software Technology 52*, 3 (2010), 304 – 323. pages 16

[79] Liskov, B. H. A design methodology for reliable software systems. In *Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part I* (New York, NY, USA, 1972), AFIPS '72 (Fall, part I), ACM, pp. 191–199. pages 7

[80] Magnenat, S., Rétornaz, P., Bonani, M., Longchamp, V., and Mondada, F. ASEBA: A Modular Architecture for Event-Based Control of Complex Robots. *IEEE/ASME Transactions on Mechatronics 16*, 2 (April 2011), 321–329. pages 24

[81] Mark Pedley. Tilt sensing using a three-axis accelerometer. https://www.nxp.com/files/sensors/doc/app_note/AN3461.pdf, 2013. pages 42

[82] Maxim Integrated Products, Inc. Using the Simple Serial Interface for Embedded Measurement Devices: 78M6610+LMU, 78M6610+PSU, and the MAX78630+PPM. https://www.maximintegrated.com/en/app-notes/index.mvp/id/5947, 2015. pages 123, 139

[83] McLurkin, J., McMullen, A., Robbins, N., Habibi, G., Becker, A., Chou, A., Li, H., John, M., Okeke, N., Rykowski, J., Kim, S., Xie, W., Vaughn, T., Zhou, Y., Shen, J., Chen, N., Kaseman, Q., Langford, L., Hunt, J., Boone, A., and Koch, K. A robot system design for low-cost multi-robot manipulation. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Sept 2014), pp. 912–918. pages 5

[84] Medeiros, A. A. D. A survey of control architectures for autonomous mobile robots. *Journal of the Brazilian Computer Society 4* (04 1998). pages 11

[85] Medvidovic, N., Tajalli, H., Garcia, J., Krka, I., Brun, Y., and Edwards, G. Engineering heterogeneous robotics systems: A software architecture-based approach. *Computer 44*, 5 (May 2011), 62–71. pages 11

[86] MIT. FabLab Leuven. http://www.fablab-leuven.be, 2011. Last visited August 2016. pages 6

[87] Molloy, D. *Exploring BeagleBone: Tools and Techniques for Building with Embedded Linux*. Wiley, 2014. pages 4

[88] Molloy, D. *Exploring Raspberry Pi: Interfacing to the Real World with Embedded Linux*. Wiley, 2016. pages 4

[89] Morbach, J., Wiesner, A., and Marquardt, W. OntoCAPE - a (re)usable ontology for computer-aided process engineering. *Computers & Chemical Engineering 33*, 10 (2009), 1546 – 1556. Selected Papers from the 18th European Symposium on Computer Aided Process Engineering (ESCAPE-18). pages 17

[90] Nilsson, N. J. *Principles of Artificial Intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1980. pages 8

[91] NISO. *Understanding metadata*. National Information Standards Organization, 2004. ISBN 1-880124-62-9. pages 72

[92] Nugraha, M. B., Ardianto, P. R., and Darlis, D. Design and implementation of RFID line-follower robot system with color detection capability using fuzzy logic. In *Control, Electronics, Renewable Energy and Communications (ICCEREC), 2015 International Conference on* (Aug 2015), pp. 75–78. pages 7

[93] Oberstar Consulting. *Fixed-Point Representation & Fractional Math*, 8 2007. Rev. 1.2. pages 95

[94] Object Management Group (2011). Unified Modeling Language (UML) superstructure specification, Version 2.4.1, 2011. pages 8

[95] Object Management Group (2015). Systems Modeling Language (SysML), Version 1.4, 2015. pages 11

[96] OpenCV Dev Team. How to use background subtraction methods. http://docs.opencv.org/3.0-beta/doc/tutorials/video/background_subtraction/background_subtraction.html. pages 101

[97] Ousterhout, J. K., Jones, K., Foster-Johnson, E., Fellows, D., Griffin, B., and Welton, D. *Tcl and the Tk Toolkit*, 2 ed. Addision-Wesley Professional Computing Series. Addison-Wesley, Upper Saddle River, New Jersey, 2010. pages 10

[98] Papp, Z., Brown, C., and Bartels, C. World modeling for cooperative intelligent vehicles. In *Intelligent Vehicles Symposium, 2008 IEEE* (June 2008), pp. 1050–1055. pages 3

[99] Paschke, A., Vincent, P., Alves, A., and Moxey, C. Tutorial on advanced design patterns in event processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems* (New York, NY, USA, 2012), DEBS '12, ACM, pp. 324–334. pages 24

[100] Petri, C. A. Kommunikation mit Automaten. Dissertation, Schriften des IIM 2, Rheinisch-Westfälisches Institut für Instrumentelle Mathematik an der Universität Bonn, Bonn, 1962. pages 10

[101] Pingree, P. J., Scharenbroich, L. J., Werne, T. A., and Hartzell, C. Implementing Legacy-C Algorithms in FPGA Co-Processors for Performance Accelerated Smart Payloads. In *Aerospace Conference, 2008 IEEE* (March 2008), pp. 1–8. pages 97

[102] Ptolemaeus, C., Ed. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. pages 23

[103] Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software* (2009). pages 14

[104] Radestock, M., and Eisenbach, S. Coordination in evolving systems. *Trends in Distributed Systems CORBA and Beyond* (2005), 162–176. pages 17, 22

[105] Rajkumar, R., Gagliardi, M., and Sha, L. The real-time publisher/subscriber inter-process communication model for distributed real-time systems: design and implementation. In *Proceedings Real-Time Technology and Applications Symposium* (May 1995), pp. 66–75. pages 49

[106] Raskin, J.-F. *An Introduction to Hybrid Automata*. Birkhäuser Boston, Boston, MA, 2005, pp. 491–517. pages 10

[107] Richards, M. *Software Architecture Patterns*. O'Reilly Media, Inc., 2015. pages 24, 47, 132

[108] Sariff, N., and Buniyamin, N. An overview of autonomous mobile robot path planning algorithms. In *2006 4th Student Conference on Research and Development* (June 2006), pp. 183–188. pages 5

[109] Savage, J. E. *Models of Computation: Exploring the Power of Computing*, 1st ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. pages 23

[110] Schamai, W., Fritzson, P., Paredis, C. J., and Pop, A. Towards unified system modeling and simulation with modelicaml: Modeling of executable behavior using graphical notations. In *Proc. 7th Modelica Conf.* (Sep. 2009). Como, Italy. pages 13

[111] Schlett, M. Trends in embedded-microprocessor design. *Computer 31*, 8 (Aug 1998), 44–49. pages 4

[112] Schmid, M., Apelt, N., Hannig, F., and Teich, J. An image processing library for C-based high-level synthesis. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)* (Sept 2014), pp. 1–4. pages 97

[113] Schwartz, M., and Manickum, O. *Programming Arduino with LabVIEW*. Packt Publ., Birmingham, 2015. pages 12

[114] Scioni, E. *Online Coordination and Composition of Robotic Skills: Formal Models for Context-aware Task Scheduling.* PhD thesis, Departement Werktuigkunde, Faculty of Engineering, KU Leuven, April 2016. Bruyninckx, Herman (supervisor). pages 17

[115] Scioni, E., Huebel, N., Blumenthal, S., Shakhimardanov, A., Klotzbücher, M., Garcia, H., and Bruyninckx, H. Hierarchical Hypergraphs for Knowledge-centric Robot Systems: a Composable Structural Meta Model and its Domain Specific Language NPC4. *JOSER 7*, 1 (2016), 55–74. pages 9, 17, 18, 31, 136

[116] Shakhimardanov, A., Hochgeschwender, N., Reckhaus, M., and Kraetzschmar, G. K. Analysis of software connectors in robotics. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Sept 2011), pp. 1030–1035. pages 11

[117] Shvets, A. *Design Patterns Explained Simply.* sourcemakeing.com, 2015. pages 16

[118] Skalicky, S., Wood, C., Lukowiak, M., and Ryan, M. High level synthesis: Where are we? a case study on matrix multiplication. In *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)* (Dec 2013), pp. 1–7. pages 102

[119] Soetens, P. *A Software Framework for Real-Time and Distributed Robot and Machine Control*. PhD thesis, Departement Werktuigkunde, Faculty of Engineering, KU Leuven, May 2006. Bruyninckx, Herman (supervisor). pages 19, 57, 133

[120] Soliman, M., Abiodun, T., Hamouda, T., Zhou, J., and Lung, C. H. Smart home: Integrating internet of things with web services and cloud computing. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science* (Dec 2013), vol. 2, pp. 317–320. pages 2

[121] Sunehra, D., Bano, A., and Yandrathi, S. Remote monitoring and control of a mobile robot system with obstacle avoidance capability. In *Advances in Computing, Communications and Informatics (ICACCI), 2015 International Conference on* (Aug 2015), pp. 1803–1809. pages 7

[122] Tsumugiwa, T., Kamiyoshi, A., Yokogawa, R., and Shibata, H. Position-detecting device for measurement of human motion in human-machine interaction. In *2007 IEEE/ASME international conference on advanced intelligent mechatronics* (Sept 2007), pp. 1–6. pages 5

[123] TSXperts. Arduino Compatible Compiler for Labview. https://www.tsxperts.com/arduino-compatible-compiler-for-labview/, 2015. pages 12

[124] Vacavant, A., Chateau, T., Wilhelm, A., and Lequièvre, L. A benchmark dataset for outdoor foreground/background extraction. In *Proceedings of the 11th International Conference on Computer Vision - Volume Part I* (Berlin, Heidelberg, 2013), ACCV'12, Springer-Verlag, pp. 291–300. pages 101

[125] Van Parys, R., and Pipeleers, G. Online distributed motion planning for multi-vehicle systems. In *Proceedings of the 2016 European Control Conference* (July 2016), pp. 1580–1585. pages xviii, 51, 52

[126] Vandael, S., Boucké, N., Holvoet, T., De Craemer, K., and Deconinck, G. Decentralized coordination of plug-in hybrid vehicles for imbalance reduction in a smart grid. In *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 2* (Richland, SC, 2011), AAMAS '11, International Foundation for Autonomous Agents and Multiagent Systems, pp. 803–810. pages 4

[127] Vanthienen, D., Klotzbücher, M., and Bruyninckx, H. The 5C-based architectural Composition Pattern: lessons learned from re-developing the iTaSC framework for constraint-based robot programming. *JOSER 5* (2014), 17–35. pages 17, 19, 21, 57, 70, 133

[128] Vukov, M. *Embedded Model Predictive Control and Moving Horizon Estimation for Mechatronics Applications.* PhD thesis, Stadius Center for Dynamical Systems, Signal Processing and Data Analytics, Department of Electrical Engineering (ESAT), Faculty of Engineering Science, April 2015. Diehl, Moritz (supervisor). pages xviii, 50, 51

[129] W3C Recommendation. State Chart XML (SCXML): State Machine Notation for Control Abstraction. http://www.w3.org/TR/scxml/, 2015. pages 59

[130] Wiener, N. *Cybernetics, Second Edition: or the Control and Communication in the Animal and the Machine.* The MIT Press, 1965. pages 2

[131] Wiki of Embedded Linux. Beagleboard: BeagleBone Capes. http://elinux.org/Beagleboard:BeagleBone_Capes, 2016. pages 4

[132] Wiki of Embedded Linux. RPi Expansion Boards. http://elinux.org/RPi_Expansion_Boards, 2016. pages 4

[133] Winkler, J., Bartels, G., Mösenlechner, L., and Beetz, M. Knowledge Enabled High-Level Task Abstraction and Execution. *First Annual Conference on Advances in Cognitive Systems 2*, 1 (December 2012), 131–148. pages 7, 132

[134] Xilinx Inc. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, 6 2011. Rev. 5.0. pages 95

[135] Xilinx Inc. *Spartan-3 FPGA Family Data Sheet*, 6 2013. Rev. 3.1. pages 95

[136] Xilinx Inc. *Vivado Design Suite User Guide: High-Level Synthesis*, 11 2015. Rev. 2015.4. pages 96, 102

[137] Xu, Y., Shuang, K., Jiang, S., and Wu, X. FPGA Implementation of a Best-Precision Fixed-Point Digital PID Controller. In *2009 International Conference on Measuring Technology and Mechatronics Automation* (April 2009), vol. 3, pp. 384–387. pages 100

[138] Zhang, L. *Applying System of Systems Engineering Approach to Build Complex Cyber Physical Systems.* Springer International Publishing, Cham, 2015, pp. 621–628. pages 2

[139] Zhang, L., Slaets, P., and Bruyninckx, H. An open embedded industrial robot hardware and software architecture applied to position control and visual servoing application. *International Journal of Mechatronics and Automation 4*, 1 (2014), 63–72. pages 88

[140] Zhao, W., Kim, B. H., Larson, A. C., and Voyles, R. M. FPGA implementation of closed-loop control system for small-scale robot. In *ICAR '05. Proceedings., 12th International Conference on Advanced Robotics, 2005.* (July 2005), pp. 70–77. pages 4

Publications

Articles in Internationally Reviewed Academic Journals

- Zhang, L., Slaets, P., Bruyninckx, H. (2014). An Open Embedded Industrial Robot Hardware and Software Architecture Applied to Position Control and Visual Servoing Application. International Journal of Mechatronics and Automation, 4(1), 63-72.

Papers at International Scientific Conferences and Symposia, Published in Full in Proceedings

- Zhang, L., Slaets, P., Bruyninckx, H. (2012). An Open Embedded Hardware and Software Architecture Applied to Industrial Robot Control. Proceedings of 2012 IEEE International Conference on Mechatronics and Automation. Chengdu, China, 5-8 August, 2012 (pp. 1822-1828) IEEE.
- Zhang, L., Slaets, P., Bruyninckx, H. (2014). An FPGA based architecture for concurrent system design applied to human-robot interaction applications. Proceedings of the 21st ISPE International Conference on Concurrent Engineering. Beijing, 8-11 September, 2014 (pp. 555-563) IOS Press.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF MECHANICAL ENGINEERING
PRODUCTION ENGINEERING, MACHINE DESIGN AND AUTOMATION DIVISION
Celestijnenlaan 300 box 2420
B-3001 Heverlee, Belgium
lin.zhang@mech.kuleuven.be